TURING

# 成长计划

## 系统程序员

李先静 编著

结合代码详细讲解程序开发方法

汇集丰富的软件开发思想

CSDN专家全新力作



人民邮电出版社
POSTS & TELECOM PRESS

# 系统程序员成长计划

在学习程序开发的过程中，你是否总是为自己遇到的一些问题头疼不已，你是否还在为写不出代码而心急如焚？作为软件开发人员，你是否时时为自己如何成为一名合格的程序员而困惑不已？没关系，本书将为你排忧解难。

这是一本介绍系统程序开发方法的书。书中结合内容详尽的代码细致讲述了不少底层程序开发基础知识，并在逐步深入的过程中介绍了一些简单实用的应用程序，最后还讲述了一些软件工程方面的内容，内容全面，语言生动，尤其适合初涉系统程序开发的人来读，有利于他们成长为更加专业的程序员。

虽然本书以“系统程序员”为名，但书中所蕴含的软件开发思想和方法也同样适用于其他的软件开发领域。各种软件开发人员、相关专业的在校学生以及软件开发爱好者也都不妨读读本书，来分享作者多年来在学习和实践中所总结的开发方法与所领悟的开发思想。

---

相关链接

《系统程序员成长计划》的综合练习项目，嵌入式 GUI FTK（Funny Toolkit）

网站：http://code.google.com/p/ftk/

邮件列表：https://groups.google.com/group/funnytoolkit/

书中代码下载地址：http://limodev.cn/download/cdrom.tar.gz

封面设计：于洋

本书相关信息请访问：图灵网站 http://www.turingbook.com
读者/作者热线：(010)51095186
反馈/投稿/推荐信箱：contact@turingbook.com

分类建议 计算机

人民邮电出版社网址：www.ptpress.com.cn

人民邮电出版社 POSTS & TELECOM PRESS 图灵原创

ISBN 978-7-115-22401-9

9 787115 224019 >

ISBN 978-7-115-22401-9

定价：45.00元

# 成长计划

## 系统程序员

李先静 编著



人民邮电出版社
北 京

**图书在版编目（CIP）数据**

系统程序员成长计划 / 李先静编著. -- 北京 : 人民邮电出版社, 2010.4
ISBN 978-7-115-22401-9

Ⅰ. ①系… Ⅱ. ①李… Ⅲ. ①软件开发 Ⅳ. ①TP311.52

中国版本图书馆CIP数据核字(2010)第031151号

## 内 容 提 要

本书以生动的语言和丰富的代码示例，运用一些相对简单的例子分析开发系统程序中可能遇到的各种问题。作者把数年的开发经验和阅读大量书籍的体会，结合他在培训新员工过程中所积累的培养方法，融会贯通在这 12 章的内容中。书中介绍了链表、数组、栈、队列和散列表等基础数据结构，也介绍了并发、同步和内存管理等系统程序中常需注意的问题，还讲解了文本处理器等具体应用程序的设计方法。

本书是初涉系统程序开发领域的人不可多得的一本参考书。书中体现的思想对于其他各种软件开发人员、相关专业的在校学生以及软件开发爱好者都有启发意义。

**系统程序员成长计划**

◆ 编　　著　李先静
责任编辑　傅志红
执行编辑　傅尔也

◆ 人民邮电出版社出版发行　　北京市崇文区夕照寺街14号
邮编　100061　　电子函件　315@ptpress.com.cn
网址　http://www.ptpress.com.cn
中国铁道出版社印刷厂印刷

◆ 开本：800×1000　1/16
印张：17.5
字数：413千字　　2010年4月第1版
印数：1－3 000册　　2010年4月北京第1次印刷

ISBN 978-7-115-22401-9

定价：45.00元

**读者服务热线：(010)51095186　印装质量热线：(010)67129223**
**反盗版热线：(010)67171154**

## 李先静

CSDN 开源专家，有着十年 Linux 开发经验、五年手机开发经验。擅长嵌入式程序员培训，软件架构设计和技术写作。近几年负责 Broncho Linux 智能手机项目，致力于基于 Linux 的嵌入式系统的学习和研究。其 CSDN 博客（http://blog.csdn.net/absurd）连续三年被 CSDN 提名为最有价值的技术博客（MVB），他先后发表了近 500 篇技术类博文，博客文章被各大技术网站转载。在《程序员》杂志上发表过多篇文章。

站在巨人的肩上

Standing on Shoulders of Giants


www.turingbook.com

# 序

## 写作背景

在经历过几个大型的、失败的项目之后，我终于认识到了：没有什么比高素质的程序员更能决定项目的成功。无论采用什么开发过程，什么编程语言和开发工具，离开了高素质的程序员，都是白费力气。毫无疑问，人是软件开发中最重要的因素。但并非每个人都重要，也不是什么样的人都重要，在软件开发中，只有那些高素质的程序员和那些对项目有突出贡献的人才是重要的。

不过高素质的程序员并不多见，所以从我开始带人起，就一直在思考团队成员培养的问题。我做过很多尝试，从小组内学习到整个部门一起上大课，最后又回到对个人做单独的辅导；从通过Code Review（代码评审）做现场教育到制定一个宏伟的培训计划，最后又回到一个朴素的培训过程。其中遇到了很多问题，开始是培训不够系统，效果不甚理想，后来又因为计划过于“宏伟”而无法实施，等到最后形成一个朴素的、切实可行的培训方案，已经经过了好几年时间，直到去年，整个计划才趋于完善。我把这个培训计划称为系统程序员成长计划，而这正是我在本书中所要介绍的。

培训内容不是来源于某本书，毕业八年来，我坚持不懈地阅读有关书籍，所读过的300多本不同类型的著作装满了家中的7个大储物箱，而这些著作囊括了大部分经典的IT图书。当然培训的内容也不是全部源于书本，这几年我在开发开源软件的过程中所收获的感悟和积累的经验也融入其中。我的培训计划并不是要阐述什么高深的道理，相反，我这本书主要是针对应届毕业生和业余爱好者写的，目的就是为了让初学者进阶为一个专业的程序员。

为什么把这个培训计划叫做“系统程序员成长计划”，而不是“程序员成长计划”呢？程序员的范围太广了，虽然软件开发有很多相似之处，但是隔行如隔山，比如对于目前炙手可热的Web开发，我完全是外行。为了不造成“想什么都讲一点，结果是什么都没有讲清楚”的尴尬，我得把培训计划限定在我熟悉的范围之内。而所谓系统程序员，是指从事操作系统内核、DBMS、GUI系统、基础函数库、应用程序框架、编译器和虚拟机等基础软件开发的程序员。不过虽说这个培训计划叫“系统程序员成长计划”，其实这些内容同样适用于桌面软件和智能手机软件开发，

对其他软件开发也多少会有一些启发作用。

第一次在温伯格的《咨询的奥秘》[①]中看到草莓酱定律和果酱定律[②]时，我觉得非常有意思。当然《系统程序员成长计划》也无法脱离草莓酱定律的魔法，利用本书所讲的内容，我手把手地教了十多个同事，取得了良好的效果。但当有成百上千的读者读这些文章时，我不敢期望有同样的效果。不过在果酱定律的鼓励下，我相信本书中至少有部分内容的价值不会因为读者群的增大而消失，所以我最终决定写这本书，来分享我这些年来所积累下的经验。

## 中心思想

软件开发的困难在哪里？对于这个问题，不同的人有不同的答案，同一个人在不同职业阶段也会有不同的答案。作为一个系统程序员来说，我认为软件开发有两大难点。

一是控制软件的复杂度。软件的复杂度越来越高，而人类的智力基本保持不变，如何以有限的智力去控制无限膨胀的复杂度？在经历过几个大型项目，也分析过不少现有的开源软件后，我得出一个结论：单个难题和技术细节我们总是可以搞定的，而所有这些问题出现在一个项目中时，其呈指数增长的复杂度往往让我们束手无策。

二是隔离变化。用户需求在变化，应用环境在变化，新技术不断涌现，所有这些都要求软件开发能够射中移动的目标。即使是开发基础平台软件，在超过几年时间的开发周期之后，需求的变化也是相当惊人的。需求变化并不可怕，关键在于变化对系统的影响，有时这种变化会牵一发而动全身，一点小小的变化都可能对系统造成致命的影响。

为了解决这两个问题，方法学家们几十年来不断努力，他们改进或发明软件的开发过程和设计方法。系统程序员所面对的基础软件通常都是复杂的大型软件，其通用性也要求能容纳更多变化，解决这两个问题也是系统程序员成长计划的主要目标。

## 文章特色

**以引导读者思考为主**。培训可以制造合格的程序员，却无法造就一流的高手。因为培训是一个相对被动的过程，很难保证学习效果（我们都知道在大学里听课的效果），所以我不希望本书被视作一本单纯的培训教材，我们要做到变被动为主动，最大限度地提高学习的效果。大多数情况下，我会先提出问题让读者去思考，让读者尝试自行解决问题。能不能解决这个问题其实并不重要，重要的是在思考中提升自己。如果读者在一定时间内找不到解决问题的方法，本书也提供

① 已由清华大学出版社出版。

② 草莓酱定律：面积涂得越大，酱就越薄。它说明如果被过分引申，任何深刻的寓意都会被减弱。果酱定律：只要还有颗粒，酱就永远都不会被涂抹得过薄。说明寓意不会因为过分引申而消失。

了专业程序员的参考解决方案（或许不是最优的）。

**以简单的例子讲述复杂的设计方法**。我曾经制定过一个宏伟的培训计划，但不幸的是这个计划并没有带来成功的结果，原因很简单：我忘记了我在学会走路之前也曾艰难地爬行过。这次我吸取了教训，用简单的示例来讲述复杂的设计方法，而且不要求读者掌握许多背景知识。书中不会出现复杂的数据结构和算法，也不会引入大型软件来唬人。既包含足够的挑战，不会让读者感到乏味；又一切尽在掌握之中，不会让读者因为挫折而打击积极性。

**技术能力与工作态度并重**。古人云："德才兼备真君子。"同样，一流的程序员也应该是德才兼修的。当我手把手教别人的时候，我希望他不仅能学会我讲的知识点，更希望他能对我的工作态度和作为程序员的道德素养有所感悟。虽然有些东西只可意会不可言传，但我仍希望大家能成为德才兼备的程序员。

## 读者群

本书主要是针对初学者写的，这里所说的初学者，包括在校学生、应届毕业生和其他业余爱好者。拿我面试过的应届毕业生来说吧，他们大多数并不具备工作所需要的编程能力，只是对基本理论多少有一些了解。本书中的文章就是为需要提升编程能力的初学者们量身定制的。书中的内容经历了十余人的实践，取得过令人满意的效果：大多数参与过我的培训的人最开始可能连一行代码都写不出，但到了培训结束时，他们一般都能独立开发/维护一些有着几千行代码的小模块。不过学习效果还是要看个人的领悟能力和努力程度，但不管怎样，只要读完这些文章，你都能从中取得不少收获。

## 如何使用本书

温伯格说过，医生的药方包括需要服用的药物和服药的方法，两者缺一不可。同样的教材，如果使用不同的学习方法，最终效果也有很大差别。那么该如何学习本书中的文章呢？我建议大家先自己想办法去解决文中提出的问题，在思考的过程中可以自己查阅资料，至少经过两三个小时的思考之后，再继续阅读下去，最后再按学到的方法自己独立地将程序写一遍。要记住：**学习编程一定要多写多练，否则效果会大打折扣。**

Enjoy it!

# 致　谢

感谢我的上司老魏（魏政群），没有他的支持，我不可能写那么多BLOG，没有他的信任，我不可能在broncho团队推行这套培训课程。感谢broncho团队同事的支持，大部分同事都参加了这套培训课程，他们的反馈让我能不断完善这套课程。

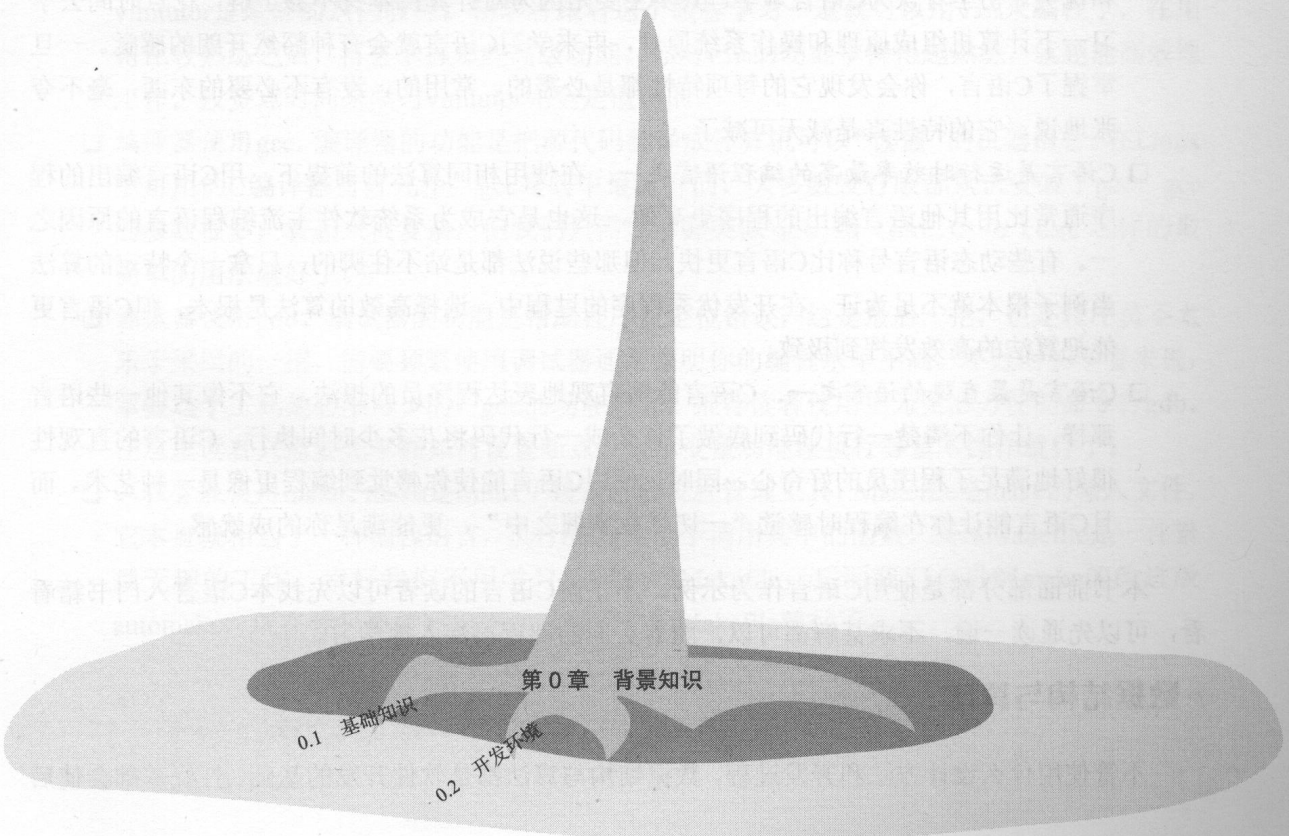
感谢我的老婆欢欢，没有她无微不至的照顾，我不可能有那么多时间去写BLOG和写书，没有她的鼓励支持，我不可能坚持下来。感谢岳父岳母，如果不是他们精心地照顾我的儿子，我不可能有精力去写作。感谢我的父母，他们的声音总是给我无穷的动力。

感谢网友们的支持，特别是Joey.Huang、Dig、haibin.yu和echo几位兄弟长期的支持，是你们的鼓励让我感觉在做一件有意义的事情。

# 目　录

# 第0章

# 背景知识



对于是否应该写这样一章，我犹豫了一段时间，最后考虑到本书主要是针对新手而写的，不应该对读者背景有过多要求，所以还是写了这一章介绍背景知识的内容。其实本章介绍的这些基础知识是每个程序员都必须掌握的。如果你已经了解它们，尽可以放心大胆地跳过本章。如果你是新手，那么请认真学习本章所讲述的内容。

## 0.1 基础知识

### C语言

千万不要认为C语言过时了，它始终是开源社区，特别是系统软件和嵌入式系统开发中的王者，在可以预见的未来，C语言还将持续不断地焕发出生命力。有些不了解软件开发的人也许会认为C语言不适合开发大型软件，这种看法是不对的，事实上，操作系统内核、虚拟机、数据库管理系统、图形引擎和Web服务器等大型软件几乎都是用C语言开发的。C语言其实并不适合开发小程序，相较而言，脚本语言更适用于小程序的开发。C语言能经久不衰，自有它的道理。

- C语言是最简单的语言之一。大部分编程语言在刚出现时都以其简洁而获得好评，但几乎都随着时间的推移而变得越来越复杂。不过C语言历经数十年的发展，却始终保持其简洁和优美。初学者认为C语言难学，其实主要是因为对计算机本身不够了解，花点时间去学习一下计算机组成原理和操作系统原理，再来学习C语言就会有种豁然开朗的感觉。一旦掌握了C语言，你会发现它的每项特性都是必需的、常用的，没有不必要的东西，毫不夸张地说，它的特性真是减无可减了。
- C语言是运行时效率最高的编程语言之一。在使用相同算法的前提下，用C语言编出的程序通常比用其他语言编出的程序更高效，这也是它成为系统软件主流编程语言的原因之一。有些动态语言号称比C语言更快，但那些说法都是站不住脚的，只拿一个特定的算法当例子根本就不足为证。在开发优秀程序的过程中，选择高效的算法是根本，但C语言更能把算法的高效发挥到极致。
- C语言是最直观的语言之一。C语言能够直观地表达程序员的想法，它不像其他一些语言那样，让你不清楚一行代码到底做了什么或一行代码将花多少时间执行。C语言的直观性很好地满足了程序员的好奇心。同时，使用C语言能使你感觉到编程更像是一种艺术。而且C语言能让你在编程时感觉“一切尽在掌握之中”，更能满足你的成就感。

本书前面部分都是使用C语言作为示例，不了解C语言的读者可以先找本C语言入门书籍看看，可以先通读一遍，不求甚解都可以，随着后面的课程再深入地学习。

### 数据结构与算法

不管使用什么设计方法和开发过程，数据结构与算法都是软件开发的基础。打好基础会使后

续的开发工作事半功倍。后继课程也都是这些基本数据结构和算法为中心，讲述如何用这些基本的材料构建大型系统。读者暂时无需精通数据结构和算法，可以先找本书看看，了解一下双向链表、动态数组、队列、堆、栈、散列表、排序和查找的基本原理就行了，后面我们会以这些数据结构为主题反复加以练习。

## 0.2 开发环境

本书重点讲解软件开发的基础知识，这些知识并不依赖于特定的平台和开发环境，读者可以根据自己的喜好来选择，但我们推荐读者使用下列开发环境。

- 操作系统使用Linux。Linux是最适合程序员使用的操作系统，它是开源的，有多种不同的发行版可供免费使用，而这些发行版大多默认安装有开发工具。全面学习Linux需要一本专门的书，不过即便你从来没接触过Linux，也犯不着惊慌失措，其实学习本书的内容只需要你花几个小时学会十来个常用的命令就够了，其他有关Linux的内容可以以后慢慢再学。
- 编辑器使用vim。编辑器的功能是创建源文件，也就是把我们编写的代码输入到电脑中。vim和emacs是Linux下最流行的代码编辑器，vim更容易上手，而且功能也很强大。它支持查找、剪切、替换等基本编辑功能，也支持符号跳转和代码补全等高级编辑特性。vimtutor是最好的入门教程，初学者跟着这个教程学习一遍就可以用vim来编程了，在用得比较熟练之后，再去掌握那些高级功能。你对vim的功能掌握得越熟练，就越能高效地工作，投资点时间来学习vimtutor完全是值得的。
- 编译器使用gcc。编译器的功能是把源代码翻译成计算机可以“读懂”的机器语言。在Linux下可用的C编译器有好几个，gcc是其中最流行的，大多数发行版都默认安装了gcc。gcc的参数很多，看起来很复杂，但我们只需要掌握类似 `gcc -g test.c -o test` 这样的最简单的用法就好了。
- 调试器使用gdb。调试器的功能是帮助程序员定位错误，这是最后一招，也是程序员不太乐于采用的一招，需要频繁使用调试器通常说明你的编程水平不高。不过对初学者来说，掌握这个工具是必不可少的。gdb的功能强大，推荐读者使用更为灵活方便的命令行gdb。在这里读者只需要先掌握如何设置断点、显示变量和继续执行等基本操作就行了。
- 工程管理使用make。make是Linux下最流行的工程管理工具，Makefile是make的输入文件，它本身就相当于一种编程语言，执行make相当于调用其中的函数。编写Makefile是一件繁琐无趣的工作，幸好我们不用学习如何编写Makefile，后面我们会讲解make的改进版automake，现在你只要能写出下面这种简单的Makefile就行了。

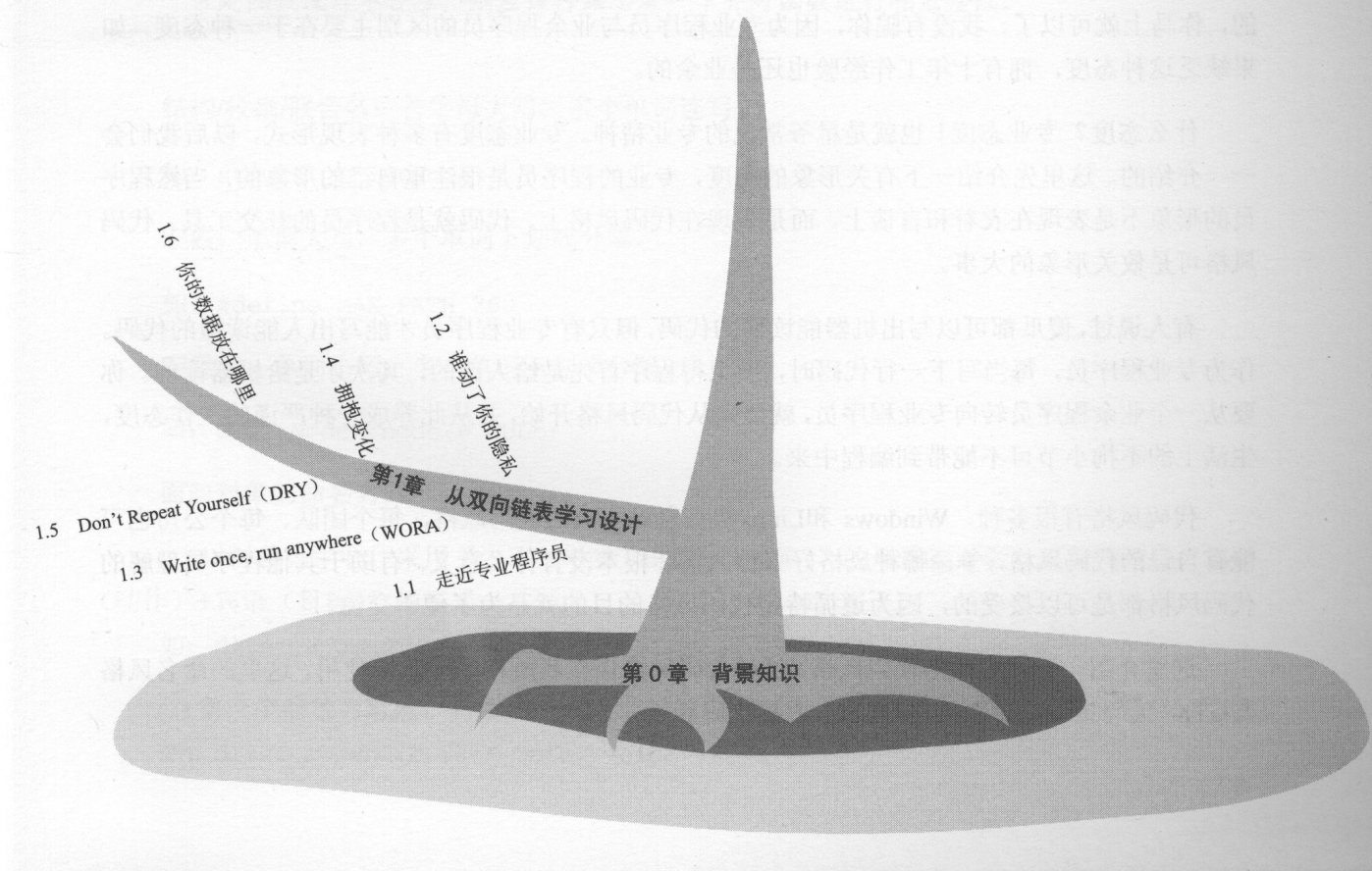
```
all:
    gcc -g test.c -o test
clean:
    rm -f test
```

在这里，你可以把all看作一个函数名，gcc -g test.c -o test是函数体（前面加tab），它的功能是将test.c编译成test，在命令行运行make all就相当于调用这个函数。clean是另外一个函数，它的功能是删除test。如果你有时间，多了解一下Makefile当然更好，如果没有时间，了解这么多也够了。

在培训初学者时，如果他从来没用过Linux，从来没有用C语言写过程序，我会给他2到4周的时间学习上述内容。如果读者也处于类似的水平，请不要急于了解后面的内容，最好先好好学习一下这里提到的知识。

# 第 1 章

# 从双向链表学习设计



# 1.1 走近专业程序员

## 需求简述

用C语言编写一个双向链表。如果你有一定的C语言编程经验，这自然是小菜一碟。有的读者可能连一个小程序都没有写过，那也不用担心，可以参考任何一本有关数据结构和C语言的书籍。先弄清楚基本概念，把书上的代码看明白，再把代码原封不动地输入到电脑里，保证编译通过，然后调试程序直到它能正常运行。重复这个过程，直到你能独立完成它为止。写第一行代码通常是很痛苦的，我培训过好几个同事，他们不是科班出身，刚开始时他们就算在电脑前坐一整天，也是连一行代码都敲不出来。其实我最早写程序时的情况也好不了多少，不过没有关系，迈出这第一步，以后的路也就好走很多了。

请先花1～3天时间完成这个任务，然后再继续往下阅读。记得多写多练，不要偷懒。

当你读到这里的时候，相信你已经独立写出了一个双向链表。恭喜你！迈出这一步可是值得庆祝的，现在你已经走在成为程序员的光明大道上了。不过你还是个业余程序员，那当然了，你才写出第一个程序呢！什么时候才能成为一个专业程序员呢？三年还是五年工作经验？其实不用的，你马上就可以了，我没有骗你，因为专业程序员与业余程序员的区别主要在于一种态度，如果缺乏这种态度，拥有十年工作经验也还是业余的。

什么态度？专业态度！也就是星爷常说的专业精神。专业态度有多种表现形式，以后我们会一一介绍的。这里先介绍一下有关形象的态度，专业的程序员是很注重自己的形象的，当然程序员的形象不是表现在衣着和言谈上，而是表现在代码风格上，代码就是程序员的社交工具，代码风格可是攸关形象的大事。

有人说过，傻瓜都可以写出机器能读懂的代码，但只有专业程序员才能写出人能读懂的代码。作为专业程序员，每当写下一行代码时，要记得程序首先是给人读的，其次才是给机器读的。你要从一个业余程序员转向专业程序员，就要先从代码风格开始，并从此养成一种严谨的工作态度，生活上的不拘小节可不能带到编程中来。

代码风格有很多种，Windows 和Linux都有自己主流的代码风格，每个团队、每个公司也可能有自己的代码风格，争论哪种风格好哪种风格坏根本没有什么意义。有助于其他程序员理解的代码风格都是可以接受的，因为遵循特定代码风格的目的就是为了便于交流。

这里介绍一下我喜欢的代码风格，这种代码风格也在我所在的团队中使用。这里的命名风格与GTK+代码相近，排版风格与Linux内核代码相近。

## 命名要展示对象的功能

**文件名**：单词小写，多个单词用下划线分隔。

如：dlist.c （这里d代表double，是通用的缩写方法。）

**注意** 文件名一定要能传达文件的内容信息，别人一看到文件名就能知道文件中放的是什么内容。把一个类的代码或者某一类代码放在一起是好的习惯，这样就很容易给文件取一个直观的名字。业余爱好者常常把很多没关系的代码糅到一个文件中，结果造成代码杂乱无章，也很难给它取一个恰当的名字。

**函数名**：单词小写，多个单词用下划线分隔。

如：find_node

**注意** 一个函数只完成单一功能。不要用代码的长度来衡量是否要把一段代码独立成一个函数。即使只有几行代码，只要这些代码完成的是一项独立的功能，都应该将其写为一个单独的函数，而函数名要能够直观地反应出它的功能。如果在给函数起名时遇到了困难，通常是函数设计不合理，则应该仔细思考一下并对函数进行相应修改。

**结构/枚举/联合名**：首字母大写，多个单词连写。

如：struct _DListNode;

**宏名**：单词大写，多个单词下划线分隔。

如：#define MAX_PATH 260

**变量名**：单词小写，多个单词下划线分隔。

如：DListNode* node = NULL;

**面向对象的命名方式**

(1) 以对象为中心，采用“主语（对象）+谓语（动作）”的形式来命名，取代传统的“谓语（动作）+宾语（目标）”的形式。

如：dlist_append

(2) 第一个参数为对象，并用thiz命名。

如：dlist_append(DList* thiz, void* value);

(3) 对象有自己的生命周期，因此都有相应的创建和销毁函数。

## 排版布局要美观大方

### 合理使用空行

(1) 函数体之间用空行分隔。

(2) 结构/联合/枚举声明用空行分隔。

(3) 不同功能的代码块之间用空行分隔。

(4) 将功能类似的代码（如宏定义、类型定义、函数声明和全局变量）放在一起，和其他部分用空行分隔。

(5) 使用空行时，一行就够了，不要使用连续多个空行，那样会让人感觉代码段空荡荡的。

### 合理使用空格

(1) 等号两边用空格。

如：`int a = 100;`

(2) 参数之间用空格。

如：`test(int a, int b, int c)`

(3) 语句末的分号与前面内容不要加空格。

如：`test(a, b, c);`

(4) 其他能让代码更美观的地方。

### 合理使用括号

(1) 用括号分隔子表达式，不要只靠默认优先级来判断。

如：`((a && b) || (c && d))`

(2) 用括号分隔`if/while/for`等语句的代码块，那怕代码只有一行。

如：

```
if(a > b)
{
    return c;
}
```

### 合理的缩进方式

每一级都正常缩进，用tab缩进取代空格缩进（Linux内核源代码也遵循此规则）。用空格缩进的目的是防止代码因编辑器的tab宽度不同而变乱，这个担心现在是多余的了，代码编辑器都

支持tab宽度设置了。如果代码缩进的层次太多（比如超过三层），则可能是代码设计上出了问题。

如：

```
if(a > b)
{
    for(i = 0; i < 100; i++)
    {
        ...
    }
}
```

**遵从团队的习惯**

这一点是最重要的，一个团队就要有一个团队的样子，不管你的水平有多高，遵循团队的规则是一个程序员的基本素养。如果团队的规则确实不好，大家应该一起完善它。

做到这一点，你已经离成为专业程序员这个目标更近一步了，重新做一遍练习吧。随着后面的学习，你就可以真正走进专业程序员这个行列了。

## 1.2 谁动了你的隐私

### ▶需求简述

或许你还在欣赏应用良好代码风格重新编写的双向链表，看起来很不错吧？不过这还远远不够，专业程序员要有精益求精的精神。至于要精到什么程度，与具体需求有关：如果只是写个小程序验证一下某个想法，那么完成所需要的功能就行了；如果是开发一个基础程序库，那就要考虑更多了。侯捷[①]先生说过："学从难处学，用从易处用。"这里我们是在学习，因此一定要精到不能再精为止。在后面的几章中，我们将对这个双向链表进行持续的重构，这个过程是循序渐进的，请大家不要着急，稳扎稳打的学习才是最好的。在这一节中，我们要学习的是程序的封装性，请先思考下面几个问题。

(1) 什么是封装？
(2) 为什么要封装？
(3) 如何实现封装？

花上半小时去思考，并尝试回答上述问题，然后按照你自己的想法重写双向链表。再次强调一下，多写多练，不要偷懒。

① 侯捷，本名侯俊杰，台湾台南人，著名计算机图书译者/作者，其主要译作有《Effective C++中文版》、《STL源码剖析》和《Win32多线程程序设计》等，其著作《深入浅出MFC》深受好评，堪称经典。——编者注

## 什么是封装

人有隐私，程序也有隐私。有隐私不是什么坏事，问题是不应该让别人知道自己的隐私，否则可能会对自己造成不小的伤害，甚至会连累相关人物跟着倒霉。程序隐私的暴露，造成的不良影响不一定会泄露个人隐私那么大，但也不容小觑。封装就是要保护好程序的隐私，不该让调用者知道的事，就坚决不要暴露出来。

## 为什么要封装

总的来说，封装主要有以下两大好处（具体影响后面再说）。

隔离变化。程序的隐私通常是程序最容易变化的部分，比如内部数据结构、内部使用的函数和全局变量等，我们需要把这些代码封装起来，从而让它们的变化不会影响系统的其他部分。

降低复杂度。接口最小化是软件设计的基本原则之一，最小化的接口容易被理解和使用。封装内部实现细节，只暴露最小的接口，会让系统变得简单明了，在一定程度上降低了系统的复杂度。

## 如何封装

### 隐藏数据结构

暴露内部数据结构会使头文件看起来杂乱无章，让调用者无所适从。而且如果调用者为图一时方便，直接访问这些数据结构的成员，就会造成模块之间紧密耦合，从而给以后的修改带来困难。隐藏数据结构的方法其实很简单，如果是内部数据结构，外面完全不会引用，则直接放在C文件中就好了，千万不要放在头文件里。如果该数据结构在内外都要使用，则可以对外暴露结构的名字，而封装结构的实现细节，做法如下。

- 在头文件中声明该数据结构。

```
struct _LrcPool;
typedef struct _LrcPool LrcPool;
```

- 在C文件中定义该数据结构。

```
struct _LrcPool
{
    size_t unit_size;
    size_t n_prealloc_units;
};
```

- 提供操作该数据结构的函数。哪怕只是存取数据结构的成员，也要包装成相应的函数。

```
void* lrc_pool_alloc(LrcPool* thiz);
void lrc_pool_free(LrcPool* thiz, void* p);
```

- 提供创建和销毁函数。因为只是暴露了结构的名字，编译器不知道它的大小（所占内存空间），外部可以访问结构的指针(指针的大小是固定的)，但不能直接声明结构的变量，所以有必要提供创建和销毁函数。

这样的声明是非法的：

```
LrcPool lrc_pool;
```

应该对外提供创建和销毁函数。

```
LrcPool* lrc_pool_new(size_t unit_size, size_t n_prealloc_units);
void lrc_pool_destroy(LrcPool* thiz);
```

不过任何规则都有例外。有些数据结构纯粹是社交型的[如点（Point）和矩形（Rect）等]，为了提高性能和方便起见，常常不需要对它们进行封装。当然封装它们也不是件坏事，MFC就对它们进行了封装，是否需要封装要根据具体情况而定。

**隐藏内部函数**

内部函数通常实现一些特定的算法（如果具有通用性，应该放到一个公共函数库里），对调用者没有多大用处，但它的暴露会干扰调用者的思路，让系统看起来比实际的复杂。函数名也会污染全局名字空间，造成重名问题。它还会诱导调用者绕过正规接口走捷径，造成不必要的耦合。隐藏内部函数的做法很简单。

(1) 在头文件中，只放最少的接口函数的声明。
(2) 在C文件中，所有内部函数都加上static关键字。

**禁用全局变量**

除了使用单件模式（只允许一个实例存在）的情况外，任何时候都要禁止使用全局变量。这一点我反复地强调，但发现初学者还是经常会为了贪图方便而使用全局变量。请大家从现在开始就记住这一准则。

全局变量始终都会占用内存空间，共享库的全局变量是按页分配的，哪怕只有一个字节的全局变量也占用一个页，这样一来就会造成不必要内存空间浪费。全局变量也会给程序并发造成困难，想把程序从单线程改为多线程将会遇到麻烦。重要的是，如果调用者直接访问这些全局变量，会造成调用者和实现者之间的耦合。

在本书中，我们都是以面向对象的方式来设计和实现程序的(封装就是面向对象的主要特点之一)。为了避免不必要的概念混淆，这里先解释一下对象和类。

- 关于对象。对象就是某一具体的事物，比如一个苹果、一台电脑都是一个对象。每个对

象都是唯一的实例，两个苹果，无论它们的外观有多么相像，内部成分有多么相似，两个苹果毕竟是两个苹果，它们是两个不同的对象。对象可以是一个实物，也可以是一个概念，比如一个苹果是实物对象，而一项政策就是一个概念对象。在软件中，对象是一个运行时概念，它只存在于运行环境中。例如，程序代码中并不存在“窗口对象”这样的东西，要创建一个窗口对象，一定要让程序运行起来才行。

- 关于类。对象可能是一个无穷的集合，用枚举的方式来表示对象集合不太现实。抽象出对象的特征和功能，按此标准将对象进行分类，这就引入类的概念。类就是一类事物的统称，它实际上就是一个分类的标准，符合这个分类标准的对象都属于这个类。当然，为了方便起见，通常只需要抽取那些对当前应用来说有用的特征和功能。在软件中，类是一个设计时概念，它只存在于代码中，运行时并不存在某个类和某个类之间的交互。我们说，编写一个双向链表，实际上指的是双向链表这个类。

C语言里并没有类这个概念，我也不想因为引入这个概念让大家感到迷惑。在后面的讲述中，我不会刻意区分类和对象，我们说对象，可能是指单个对象，也可能是指对象所属的类，要根据上下文进行区分（这种区分通常是很直观的）。我并不是这种做法的首创者，见过好几本书都是这样做的，希望大家要清楚地理解这个概念问题。

好了，如果你实现的双向链表没有达到这个标准，那么请重做一遍练习吧。

## 1.3 Write once, run anywhere（WORA）

### ▶需求简述

“Write Once, Debug Everywhere”，据说这是流传于JAVA程序员中间的一句笑话，而Sun公司用来形容JAVA的跨平台性的原话是“Write once, run anywhere”（WORA）。后者描述的是一种理想状态，前者才是在讲现实情况。如果我们的双向链表可以到处运行，那就太好了。“Write once, run anywhere”是我们的目标，不过我们先要面对现实，回到双向链表上，首先要请大家思考下列问题。

(1) 专用双向链表和通用双向链表各自的特点与适用范围。
(2) 如何编写一个通用的双向链表？

花点时间思考一下，再尝试编写一个通用的双向链表，或许实现得不是那么完美，但是我们迈出了走向通用性的第一步。

### ▶专用链表和通用链表各自的特点与适用范围

专用链表在这里是指该链表的实现和调用耦合在一起，只能被一个调用者使用，而不能单

独在其他地方被重用。通用链表则相反，它具有通用性，可以在多处被重复使用。尽管通用链表相对专用链表来说有很多优越之处，不过草率地断定通用链表比专用链表好也是不公正的，因为它们都有自己的优点和适用范围。



**注意** 在本节中，为了避免读起来拗口，我把双向链表简写成链表了，希望大家不要介意。

#### 专用链表的优点

- 性能更高。专用链表的实现和调用在一起，可以直接访问数据成员，省去了包装函数带来的性能开销，可以提高时间性能。专用链表无需实现完整的接口，只要满足自己的需要就行了，生成的代码更小，因此可以提高空间性能。
- 依赖更少。自己实现不用依赖于别人。有时候你要写一个规模不大的跨平台程序，比如想在展讯手机平台和MTK手机平台上运行，虽然有现存的库可用，但你又不想把整个库移植过去，那么实现一个专用链表是不错的选择。
- 实现简单。实现专用链表时，不需要考虑在各种复杂应用情况下的特殊要求，也不需要提供完整的接口，所以实现起来比通用链表更为简单。

#### 通用链表的优点（从全局来看）

- 可靠性更高。通用链表的实现要复杂得多，复杂的东西意味着不可靠。但它是可以重复使用的，其存在的问题会随每一次重用而被发现和改正，慢慢地就形成一个可靠的函数库。
- 开发效率更高。通用链表的实现要复杂得多，复杂的东西也意味着更高的开发成本。同样因为它是可以重复使用的，开发成本会随每一次重用而降低，从整个项目来看，会大大提高开发效率。

考虑到链表是最常用的数据结构之一，很多地方都会用到它，实现通用的链表会更有价值。接下来我们要实现一个通用的链表，不过请大家记住，实现通用的链表并不是我们的目标，而是我们学习软件设计方法的手段。前面我许诺过要以简单的数据结构讲述复杂的软件设计方法，链表就是其中的载体之一。

## 如何编写一个通用的链表

编写通用链表是一项复杂的任务，不可能在这一节中把它阐述清楚，这里我们先考虑三个问题。

#### 存值还是存指针

通用链表首先要做到能够存放任何数据类型的数据，新手常见的做法是定义一个抽象数据类型，需要存放什么，就定义成什么。如：

```
typedef int Type;
typedef struct _DListNode
{
    struct _DListNode* prev;
    struct _DListNode* next;
    Type data;
}DListNode;
```

这样的链表算不上是通用的，因为你存放整数时编译一次，存放字符串时，重新定义Type再编译一次，存放其他类型同样要重复这个过程。麻烦不说，关键是没有办法同时使用多个数据类型。我们要找到一种同时可以表示不同数据类型的类型才行，有人说可以用union，但是数据类型是无穷无尽的，不可能在 union中表示它们的全部。

可行的办法有两种。

❑ *存值*

```
typedef struct _DListNode
{
    struct _DListNode* prev;
    struct _DListNode* next;
    void*  data;
    size_t length;
}DListNode;
```

存入时复制一份数据，保存数据的指针和长度。考虑到复制数据会带来性能开销，不合符C语言的风格，而且C语言中没有构造函数，实现深复制比较麻烦，所以在C语言中以这种方式实现的链表很少见。

❑ *存指针*

```
typedef struct _DListNode
{
    struct _DListNode* prev;
    struct _DListNode* next;
    void*  data;
}DListNode;
```

只保存指向对象的指针，存取效率高，是C语言中常见的做法。在存放整数时，可以把void*强制转换成整数使用，以避免内存分配(在现实中，90%以上的情况下，链表都是存放结构的)。

**让C++可以调用**

这不是一个和本书主旨密切相关的话题，因此在这里只是稍微提一下。C++中允许同名函数存在，所以编译器会对函数名重新编码。当C++代码包含C语言的头文件时，重新编码的函数名如果与C语言库中的原函数名不一致，就会造成找不到函数的情况。为了让用C语言实现的函数

可以在C++中调用，需要在头文件中加点东西才行。

```
#ifdef __cplusplus
extern "C" {
#endif
...
#ifdef __cplusplus
}
#endif
```

它表示如果在C++中调用这里的函数，编译器不能对函数名进行重新编码。

**完整的接口**

作为一个通用的链表，接口要比较完整才行，否则无法满足各种情况的需要(提供完整的接口并不违背最小接口原则)。实现具有完整接口的链表不是件容易的事，在这里你只需要先实现插入删除等基本操作就行了，后面我们会一步一步地继续扩展它的功能。

## 1.4 拥抱变化

### 需求简述

大部分初学者在编写双向链表时，为了验证相关函数是否能正常工作，都会去编写一个dlist_print函数，它的功能是在屏幕上打印出整个双向链表中的数据。客观地说，用dlist_print输出的信息来判断dlist的正确性并不是最好的办法，不过有保障程序质量的意识总是值得表扬的。当把专用双向链表演化成通用双向链表时，编写一个dlist_print就不那么简单了。这里请大家自行写一个dlist_printf函数，看看会遇到什么问题。

在专用双向链表中，dlist_printf的实现非常简单，如果里面存放的是整数，用 %d 打印，存放的是字符串，用 %s 打印。现在的麻烦在于双向链表是通用的，我们无法预知其中存在的数据类型，也就是说我们要面对数据类型的变化。怎么办呢？初学者可以参考的常用方法有以下几种。

**实现多个函数，需要哪个就用哪个**

比如实现dlist_print_int用来打印存放整数的双向链表，dlist_print_string用来打印存放字符串的双向链表等，其他类型都有自己的打印函数。

不过这种做法也有一些缺点。一是每个函数的实现方式类似，会带来大量重复的代码。二是由于数据类型的种类不确定，如果为每种数据类型都实现一个print函数，当要存放新的数据类

型时，就不得不修改dlist的实现。

**传入一个附加参数来决定如何打印**

比如传入1表示按整数方式打印，传入2表示按字符串方式打印，以此类推。

这种做法比第一种好一点，至少不会造成大量重复的代码。但是同样存在增加新类型时要修改dlist_print函数的问题。

**调用dlist的接口函数获取每一个位置的数据并打印出来**

这种方法没有前面两种方法的缺点，而且是一种相当直观的方式。但奇怪的是偏偏很少有人使用这个方法，原因可能有两个：其一是太拘泥于传统的实现方式而没有想到这一种；其二是担心性能问题，因为通过索引取值，每一次都要从头开始定位，其性能开销为O(n*n)。

其实这种方法是可以接受的，dlist_print函数只是用于辅助测试，我们并不需要太在乎它的性能开销，而且我们很少会在链表中存放成千上万的数据，因此这个函数带来的性能影响根本没有想的那样严重。

不过在这里我们要介绍一种新的方法。

dlist_print的大体框架如下。

```
DListNode* iter = thiz->first;
while(iter != NULL)
{
    print(iter->data);
    iter = iter->next;
}
```

在上面代码中，我们主要是不知道如何实现 print(iter->data); 这行代码。那么谁知道呢？很明显，调用者知道，因为调用者知道链表里面所存放的数据类型。好吧，那就让调用者来做好了，调用者在调用dlist_print时会提供一个函数给dlist_print来调用，这种回调调用者所提供函数的方法，我们可以称之为回调函数法。

调用者如何提供函数给dlist_print呢？当然是通过函数指针了。变量指针指向的是一块数据，指针指向不同的变量，则取到的是不同的数据。函数指针指向的是一段代码（即函数），指针指向不同的函数，则具有不同的行为。函数指针是实现多态的手段，多态就是隔离变化的秘诀，这里只是一个开端，后面我们会逐步地深入学习。

回到正题上，我们看如何实现dlist_print。

- 定义函数指针类型。

```
typedef DListRet (*DListDataPrintFunc)(void* data);
```

❑ 声明dlist_print函数。

```
DListRet dlist_print(DList* thiz, DListDataPrintFunc print);
```

❑ 实现dlist_print函数。

```
DListRet dlist_print(DList* thiz, DListDataPrintFunc print)
{
    DListRet ret = DLIST_RET_OK;
    DListNode* iter = thiz->first;

    while(iter != NULL)
    {
        print(iter->data);
        iter = iter->next;
    }
        return ret;
}
```

❑ 调用方法。

```
static DListRet print_int(void* data)
{
    printf("%d", (int)data);

        return DLIST_RET_OK;
}
...
dlist_print(dlist, print_int);
```

所有问题迎刃而解，是不是很简单呢？我以前写过一篇关于函数指针的BLOG，文中声称不懂函数指针就不要自称是C语言高手，现在我仍然坚持这个观点。函数指针的概念本身其实很简单，关键就在于我们能否将其灵活应用，这里只展示了一个最简单的应用，希望大家能够仔细体会一下，本书后面部分还会有大量篇幅来介绍函数指针的应用。

## 1.5 Don't Repeat Yourself（DRY）

### ▶需求简述

这里请大家自己编写程序实现下列功能。

(1) 对一个存放整数的双向链表，找出链表中的最大值。
(2) 对一个存放整数的双向链表，累加链表中所有整数。

多写多练，不要偷懒，写完之后请仔细思考一下自己的程序有没有可以改进的余地。

实现这两个函数并不是件难事，但真正写好的人并不多。初学者通常可以参考以下两种做法。

**各写一个独立的函数**

写一个dlist_find_max函数用来找出最大值，再写一个dlist_sum函数用来求和。这种做法和前面写dlist_print时所犯的错误一样，会造成重复的代码，也让dlist的实现需要随着应用环境的改变而改变。

**采用回调函数法**

细心的初学者会发现，这两个函数的实现与dlist_print的实现很类似，无非是print那行代码要换成别的功能。能想到这一点很好，不过在真正动手时，发现每个回调函数都要保存一些中间数据。大部分人选择了用全局变量来保存，这可以实现要求的功能，但违背了禁用全局变量原则。

这两个函数其实并没有太多实用价值，但是通过它们我们可以学习几点。

**不要编写重复的代码**

按传统的方法写出dlist_find_max之后，每个人都知道这个函数与dlist_print很类似，在写出dlist_sum之后，那种感觉就更明显了。在这个时候，你不应该停下自己前进的脚步，而是要自己想办法把这些重复的代码抽出来。即使因为经验所限感到这样做很困难，你也要尽全力去多思考和查资料。

写重复的代码很简单，有时甚至凭本能都可以写出来。但要想成为优秀的程序员，你就一定要克服自己的惰性，因为重复的代码造成很多问题。

- 重复的代码更容易出错。在写类似代码的时候，几乎所有人（包括我）都会选择“复制+粘贴”的方法，但使用这种方法很容易犯一些细节上的错误，如果某个地方修改不完整，那就等于给程序埋下一枚“不定时”的炸弹，说不定什么时候会给程序带来致命打击。
- 重复的代码经不起变化。无论是修正bug，还是增加新特性，你往往需要修改很多地方，如果不慎忘掉其中一处，你同样得为此付出代价。不是有这样一句电影台词么——“出来混，迟早是要还的”。对软件开发而言，情况就更严重了。大师们曾经说过，在软件中如果欠下bug，你会为此还得更多。

去除重复代码往往不是件简单的事情，它需要我们付出更多思考和更多精力，不过事实证明这绝对称得上是一笔超值的投资。但问题又来了，在这里，我们要怎样抽取这些重复的代码呢？

这三个函数无非是要遍历双向链表并做一些事情，其实我们可以提供dlist_foreach函数来遍历双向链表，至于到底要做什么，这可是千变万化的行为，我们应该通过回调函数让调用者去

做这些事情。

**任何回调函数都要有上下文**

大部分初学者虽然选择了回调函数法，却又无一例外地选择了用全局变量来保存中间数据，在这里我就不再强调全局变量的坏处了，如果你对这些坏处的印象不深刻的话，可以回过头去看看前面的内容。我想要给大家介绍的是：在这种情况下，如何避免使用全局变量。

其实方法很简单，给回调函数传递额外的参数就行了。这个参数我们称为回调函数的上下文，变量名用ctx（context的缩写）。要在这个上下文中存放什么东西呢？那得根据具体的回调函数而定，为了能保存任何数据类型，我们选择void*表示这个上下文。

下面我们看看怎么实现这个dlist_foreach。

```
DListRet dlist_foreach(DList* thiz, DListVisitFunc visit, void* ctx)
{
    DListRet ret = DLIST_RET_OK;
    DListNode* iter = thiz->first;

    while(iter != NULL && ret != DLIST_RET_STOP)
    {
        ret = visit(ctx, iter->data);

        iter = iter->next;
    }

    return ret;
}
```

在上面这段程序中，visit是回调函数，ctx就是我们说的上下文。但我要特别强调一点，ctx应该作为回调函数的第一个参数。这是为什么呢？在之前所讲过的面向对象的函数命名规则中，我们要以thiz作为函数的第一个参数，而thiz通常也就是函数的上下文。如果在这里恰好ctx==thiz，就不需要因为参数顺序不同而做转换了。

实现求和的回调函数。

```
static DListRet sum_cb(void* ctx, void* data)
{
    long long* result = ctx;
    *result += (int)data;

    return DLIST_RET_OK;
}
```

调用foreach。

```
long long sum = 0;
```

```
dlist_foreach(thiz, sum_cb, &sum);
```

是不是很简单？以后在使用回调函数时，记得多加一个ctx参数，即使暂时用不着，留着它也可以方便以后扩展。好了，请大家用类似的方法实现查找最大值的功能吧。

**只做份内的事**

我见过不少任劳任怨的程序员，别人让他做什么他就做什么，不管是不是份内的事，不管是上司要求的还是同事要求的，都来者不拒。别人说需要一个某某功能的函数，他就写一个在他的模块里，日积月累，他的模块就成了一锅“大杂烩”。我亲眼见过有程序员在系统设置和桌面两个模块里，提供很多毫不相干的函数，这些函数会造成不必要的耦合和复杂度。

在这里也是一样的，求和与求最大值并不是dlist应该提供的功能，放在dlist里面实现是不应该的。为了能实现这些功能，我们提供一种满足这些需求的机制就好了。热心肠是好的，但一定不要“管得太宽”，否则就费力不讨好了。

## 1.6 你的数据放在哪里

### ▶需求简述

对一个存放字符串的双向链表，把存放在其中的字符串转换成大写字母。

谨记，多写多练，不要偷懒。

对于初学者来说这道题有点难度，很少有人能完全做对。不过没关系，我并不是要出一道难题来难倒大家，而是要刺激大家去思考，以期达到加深学习印象的效果。有了前面两次的经验，我想应该没人会去写一个dlist_to_upper函数，大家都会调用dlist_foreach来实现。不过新的问题又出现了，初学者还是有可能犯以下几种常犯的错误。

**转换大写的方法不对**

```
char* p = str;
if(p != NULL)
{
    while(*p != '\0')
    {
        if('a' <= *p && *p <= 'z')
        {
            *p = *p - ('a' - 'A');
        }
        p++;
    }
}
```

这是我们在课本里学到的写法，但在工程中是不能这样做的。因为大小写字母在不同语言中的定义是不一样的，“a”是一个字符常量，它的值在任何时候都是97，但在不同语言中，97却不一定代表“a”。我们不能简单地认为在97（a）—122（z）之间的字符就是小写字母，而是应该调用标准C函数islower来判断，同样转换为大写应该调用toupper而不是减去一个常量。



**在双向链表中存放常量字符串，转换时出现段错误。**

```
DList* dlist = dlist_create();
dlist_append(dlist, "It");
dlist_append(dlist, "is");
dlist_append(dlist, "OK");
dlist_append(dlist, "!");
dlist_foreach(dlist, str_toupper, NULL);
dlist_destroy(dlist);
```

运行时会出现“Segmentation fault”错误。原因是“It”等字符串是常量，常量是不能被修改的。

**在双向链表中存放的是临时变量，转换后发现所有字符串都一样。**

```
char str[256] = {0};
DList* dlist = dlist_create();
strcpy(str, "It");
dlist_append(dlist, str);
strcpy(str, "is");
dlist_append(dlist, str);
strcpy(str, "OK");
dlist_append(dlist, str);
strcpy(str, "!");
dlist_append(dlist, str);
dlist_foreach(dlist, str_toupper, NULL);
dlist_foreach(dlist, str_print, NULL);
dlist_destroy(dlist);
```

运行时发现打印出几个感叹号。原因是执行dlist_append时没有复制一份，所以在dlist中存放的是同一个地址。而且这个dlist在当前函数返回后，里面保存的数据都无效了，因为这些数据指向的是临时变量。

**存放时复制了数据，但没有释放所分配的内存。**

```
DList* dlist = dlist_create();
dlist_append(dlist, strdup("It"));
dlist_append(dlist, strdup("is"));
dlist_append(dlist, strdup("OK"));
dlist_append(dlist, strdup("!"));
dlist_foreach(dlist, str_toupper, NULL);
dlist_foreach(dlist, str_print, NULL);
dlist_destroy(dlist);
```

这里看起来工作正常了，但存在内存泄露的bug。strdup调用malloc分配了内存，但没有地方去释放它们。

初学者对内存和指针只有一知半解的认识，常常犯一些连自己都莫名其妙的错误。为了避免这些不必要的错误，今天我们要学习各种数据存放的位置以及它们的特性，让初学者对编程有更进一步的认识。在程序中，数据存放的位置主要有以下几个。

**未初始化的全局变量（.bss段）**

已经记不清bss代表Block Storage Start还是Block Started by Symbol。像我这种没有和那些古董级计算机打过交道的人，终究无法理解这样怪异的名字，记不住也就不足为奇了。不过没有关系，我们不必纠结于bss究竟代表什么，而是要弄清楚bss段中都会存放些什么数据、这些数据都有什么样的特点以及我们该如何使用它们。

通俗地讲，bss段被用来存放那些没有初始化或初始化为0的全局变量。它有什么特点呢，让我们先来看看一个小程序的表现。

```
int bss_array[1024 * 1024];

int main(int argc, char* argv[])
{
        return 0;
}
# gcc -g bss.c -o bss.exe
# ls -l bss.exe
-rwxrwxr-x 1 root root 5975 Nov 16 09:32 bss.exe
# objdump -h bss.exe |grep bss
24 .bss         00400020  080495e0  080495e0  000005e0  2**5
```

变量bss_array的大小为4M，而可执行文件的大小只有5K。由此可见，bss类型的全局变量只占运行时的内存空间，而不占用文件空间。

现在大多数操作系统在加载程序时，会把所有的bss全局变量清零。但为了保证程序的可移植性，最好能手工把这些变量初始化为0，这样可以使这些变量都有个确定的初始值。

当然了，作为全局变量，在整个程序的运行周期内，bss数据是一直存在的。

**初始化过的全局变量（.data段）**

与bss相比，data段就容易理解多了，看名称就大概能知道它里面存放着数据。当然，如果数据全是0，为了优化考虑，编译器会把它当作bss处理。通俗地讲，data段被用来存放那些初始化为非0值的全局变量。那么它又有什么特点呢，我们还是先来看看一个小程序的表现。

```
int data_array[1024 * 1024] = {1};

int main(int argc, char* argv[])
{
        return 0;
}
# ls -l data.exe
-rwxrwxr-x 1 root root 4200313 Nov 16 09:34 data.exe
# objdump -h data.exe |grep \\.data
23 .data        00400020  080495e0  080495e0  000005e0  2**5
```

仅仅是把初始化的值改为非0值了，文件就变为4M多。由此可见，data类型的全局变量是既占文件空间，又占用运行时内存空间的。

同样，作为全局变量，在整个程序的运行周期内，data数据也是一直存在的。

**常量数据（.rodata段）**

rodata的意义同样明显，ro代表read only（只读），rodata就是用来存放常量数据的。关于rodata类型的数据，要注意以下几点。

(1) 常量不一定就放在rodata里，有的立即数直接和指令编码在一起，存放在代码段（.text）中。

(2) 对于字符串常量，编译器会自动去掉重复的字符串，保证一个字符串在一个可执行文件（EXE/SO）中只存在一个副本。

(3) rodata是在多个进程间共享的，这样可以提高运行空间利用率。

(4) 在有的嵌入式系统中，rodata放在ROM（或者NOR闪存芯片）里，运行时直接读取，无需加载到RAM中。

(5) 在嵌入式Linux系统中，也可以通过一种叫作XIP（就地执行）的技术直接读取常量数据，而无需加载到RAM中。

(6) 常量是不能修改的，修改常量在Linux下会出现段错误。

由此可见，把在运行过程中不会改变的数据设为rodata类型是有好处的。在多个进程间共享，可以大大提高空间利用率，甚至能不占用RAM空间。同时由于rodata在只读的内存页面中是受保护的，任何试图对它进行修改的行为都会被及时发现，这样一来还可以提高程序的稳定性。

字符串会被编译器自动放到rodata中，其他数据要放到rodata中，只需要为其加const关键字修饰即可。

**代码（.text段）**

text段存放代码（如函数）和部分整数常量，它与rodata段很相似，相同的特性我们就不重复了，主要的区别在于text段是可以执行的。

### 栈（stack）

栈是用来存放临时变量和函数参数的。将栈作为一种基本数据结构，我并不感到惊讶；将其用来实现函数调用，也是大家司空见惯的作法。直到我试图找到另外一种方式实现递归操作时，我才感叹于栈的巧妙。要实现递归操作，不用栈不是不可能，只是找不出比使用栈更优雅的方式。

尽管大多数编译器在优化时会把常用的参数或局部变量放入寄存器中，但用栈来管理函数调用时的临时变量（局部变量和参数）才是通行的做法，前者只是辅助手段而已，而且只可以在当前函数中将参数和局部变量存入寄存器，一旦调用下一层函数，这些值还是得存入栈中才行。

通常情况下，栈是向下（低地址）增长的，每向栈中PUSH一个元素，栈顶就向低地址扩展，每从栈中POP一个元素，栈顶就向高地址回退。这里有一些比较有意思的问题：在x86平台上，栈顶寄存器为ESP，那么ESP的值是在PUSH操作之前修改呢，还是在PUSH操作之后修改呢？PUSH ESP这条指令会向栈中存入什么数据呢？据说x86系列CPU中，除了286外，都是先修改ESP，再压栈的。由于286没有CPUID指令，因此有的操作系统会用这种方法检查286的型号。

要注意的是，存放在栈中的数据只在当前函数及下一层函数中有效，一旦函数返回了，这些数据也就自动释放了，继续访问这些变量会造成意想不到的错误。

### 堆（heap）

堆是最灵活的一种内存，它的生命周期完全由使用者控制。标准C提供以下几个函数来使用堆内存。

| 函数名 | 用途 |
| --- | --- |
| Malloc | 用来分配一块指定大小的内存。 |
| Realloc | 用来调整/重分配一块存在的内存。 |
| free | 用来释放不再使用的内存。 |

使用堆内存时请注意以下两个问题。

- alloc/free要配对使用。我们将内存分配了而不释放的情形称为内存泄露（memory leak），如果内存泄露的情况过多出现，迟早会造成“Out of memory”（内存不足）的错误，从而无法再成功分配内存。当然释放时也只能释放已经被分配的内存，释放无效的内存或重复释放都是不行的，会造成程序崩溃。
- 分配多少用多少。分配了100字节就只能用100字节，不管是读还是写，都只能在这个范围内，读多了会读到随机的数据，写多了会造成随机的破坏。我们将读写分配范围外的数据的情况称为缓冲区溢出（buffer overflow），这种情况是非常严重的，大部分安全问题都是由缓冲区溢出引起的。

想手工检查有无内存泄露或缓冲区溢出是很困难的，幸好有些工具可供使用，比如Linux下有valgrind，它的使用方法很简单，大家下去可以试用一下，以后每次写完程序都应该用valgrind跑一遍。



最后，我们来看看在Linux下，程序运行时空间的分配情况。

```
# cat /proc/self/maps

00110000-00111000 r-xp 00110000 00:00 0          [vdso]
009ba000-009d6000 r-xp 00000000 08:01 768759     /lib/ld-2.8.so
009d6000-009d7000 r--p 0001c000 08:01 768759     /lib/ld-2.8.so
009d7000-009d8000 rw-p 0001d000 08:01 768759     /lib/ld-2.8.so
009da000-00b3d000 r-xp 00000000 08:01 768760     /lib/libc-2.8.so
00b3d000-00b3f000 r--p 00163000 08:01 768760     /lib/libc-2.8.so
00b3f000-00b40000 rw-p 00165000 08:01 768760     /lib/libc-2.8.so
00b40000-00b43000 rw-p 00b40000 00:00 0
08048000-08050000 r-xp 00000000 08:01 993652     /bin/cat
08050000-08051000 rw-p 00007000 08:01 993652     /bin/cat
0805f000-08080000 rw-p 0805f000 00:00 0          [heap]
b7fe8000-b7fea000 rw-p b7fe8000 00:00 0
bfee7000-bfefc000 rw-p bffeb000 00:00 0          [stack]
```

每个区间都有四个属性：

r 表示可以读取；

w 表示可以修改；

x 表示可以执行；

p/s 表示是否为共享内存。

对有文件名的内存区间而言：

属性为r--p表示存放的是rodata；

属性为rw-p表示存放的是bss和data；

属性为r-xp表示存放的是text数据。

没有文件名的内存区间则表示用mmap映射的匿名空间。

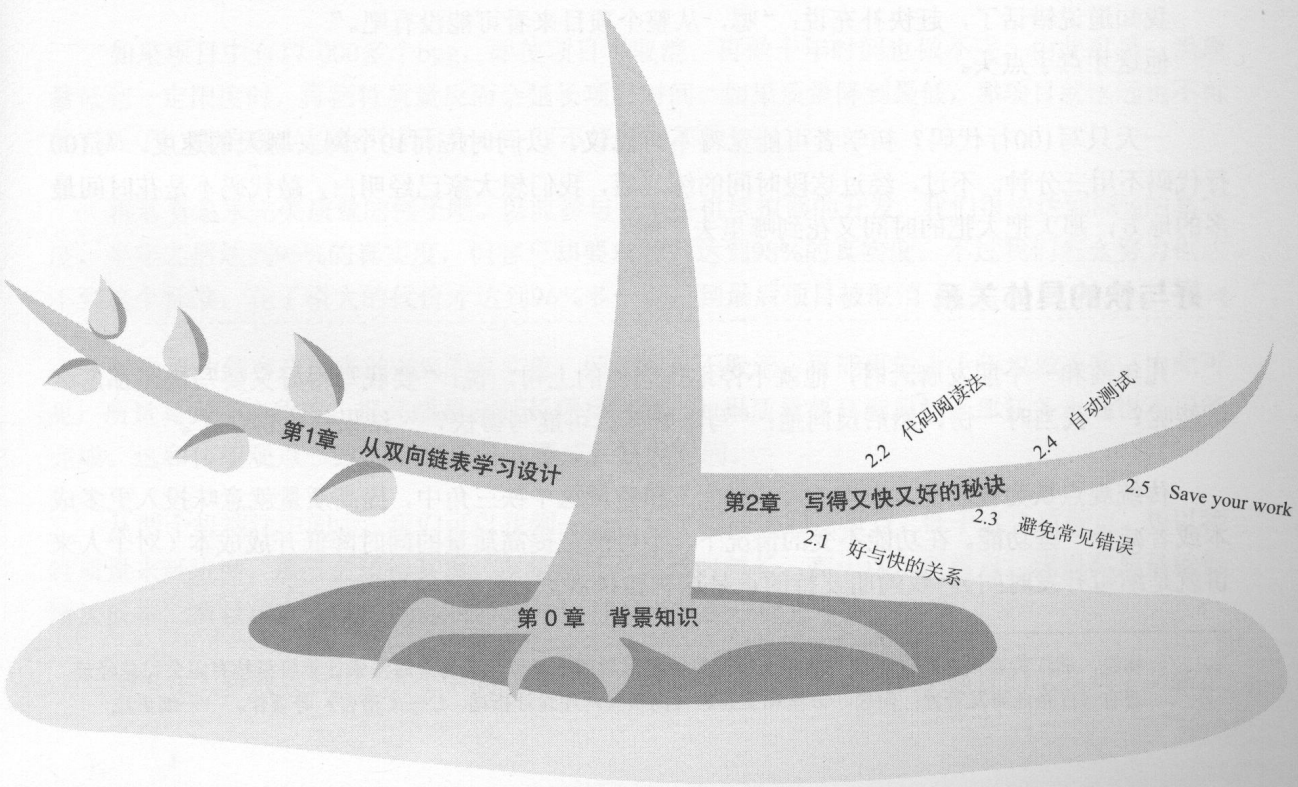
文件名为[stack]的内存区间表示是栈。

文件名为[heap]的内存区间表示是堆。

对内存的掌握是系统程序员必备的技能，希望大家多加体会。

# 第 2 章

# 写得又快又好的秘诀



## 2.1 好与快的关系

“快”是指开发效率高，“好”是指软件质量高。呵呵，写得又快又好的人就是高手了。记得这是林锐[①]博士下的定义，读他那本著名的《高质量程序设计指南——C++/C语言》时，我还是个初学者，所以对他所说的印象特别深。直到现在我仍然十分赞同他的观点，不过我这里用了个颇具“标题党”意味的标题——写得又快又好的秘诀，不过感觉倒也比较贴切。那么废话少说，请大家回顾一下这段时间的编程经验，回答下面两个问题。

**快与好是什么关系？**

写得快就不能写得好么？写得好就不能写得快么？还是写得好才能写得快呢？是不是绕晕了呢，不过这些确实是值得思考的问题。

**我们的时间花在哪里了？**

记得刚来深圳时到华为面试，面试官是我的学长。
当时他问我：“你一天能写多少行代码？”
我略加思索后回答说：“100行吧。”
他用看外行的眼光看着我，问：“真能写100行吗？”
我知道说错话了，赶快补充说：“嗯，从整个项目来看可能没有吧。”
他这才点了点头。

一天只写100行代码？初学者可能觉得不可思议，以同时应付10个网友聊天的速度，写100行代码不用三分钟。不过，经过这段时间的练习后，我们想大家已经明白，敲代码不是花时间最多的地方，那大把大把的时间又花到哪里去了呢？

### 好与快的具体关系

几年前和一个朋友聊天时，他就不停地抱怨他的上司，说：“要我写得好又要写快，那怎么可能呢？”我当时一愣，然后反问他：“写不好怎么可能写得快？”结果他也愣住了。

传统观点认为在功能、成本（人×时间）和质量这个铁三角中，提高质量就意味投入更多成本或者减少一些功能。在功能不变的情况下，不可能在提高质量的同时降低开成成本（对个人来讲就是缩短开发时间）。我的朋友持的正是这种传统观点。

① 林锐，浙江黄岩人，2000年获得浙江大学计算机图形学博士学位，现为上海漫索计算机科技有限公司总经理，著有《IT企业研发管理：问题、方法和工具》、《高质量程序设计指南—C++/C语言》等著作。——编者注

而根据我在实际项目中获得的经验来看，结论恰恰相反。每次我想以牺牲质量来加快速度的时候，结果反倒是让我花费更多的时间，甚至可能到最后因为搞不定而放弃。有了多次这样的教训之后，我决定把每一步都做好，从开发的前期来看，我花的时间比别人多一点，但从全局来看，我反而能以几倍于别人的速度完成任务。时间长了，我形成了这样的观念：只有写得好才可能写得快。



两种观点截然相反，所以我们都愣了。虽然我相信我的经验没有错，但传统的铁三角定律可是大师们总结出来的，也应该不会出错。那究竟是怎么回事呢？我开始到处查找资料，但是没有一个人支持我的观点。我又不想这样放弃，后来我用了一个简单的办法进行推理，结果证明两个观点都有各自的适用范围。

这个推理过程很简单，却把两种观点推向极端。

先看看以牺牲质量来追求进度的例子。就拿我以前参加过的两个失败的大项目来说吧，其中一个项目的bug总数达到17 000多个，结果是花了近三年时间进行开发后项目被取消；另一个项目的bug总数也超过10 000个，三年之后好歹是将尚带着很多bug的产品给发布了，但结果可想而知，产品很快从市场上消失了。这两个项目在开始阶段都制定了极其可笑的项目计划，为了赶在天方夜谭般的最后期限前完成工作，都采用了牺牲质量的方式来提高开发速度，前期进展都很“顺利”，基本功能点很快就完成了，但是项目马上陷入了无止境的debug之中，开发人员的士气一下跌到谷底，管理层开始暴跳如雷。

如果项目中有17 000多个bug，即使项目不取消，再做十年时间也做不完。由此可见：当质量低到一定限度时，再牺牲质量反而会延长项目时间，如果质量降到最低，那项目就永远也不可能完成了。我的观点与此不谋而合——写不好就写不快。

再看看追求完美质量的例子吧。以前参与一个手机模拟器的开发，我们很快达到88%的真实度，半年之后达到95%的真实度，但客户却要求我们达到98%的真实度。不过我们怎么努力也达不到这个标准，花了极大的代价才达到96%多一点，到最后项目被取消了。

如果要达到客户要求的98%的真实度，即使项目不取消，可能再做上十年也做不完。由此可见：质量高到一定程度，提高质量会延长项目时间，如果质量要高到最高，那任务永远也不可能完成。这和传统观点一致，提高质量就要延长开发时间。

从两个极端往中间走，我们可以找到一个达到平衡的中间质量点。低于这个质量点，想以牺牲质量来赶进度，那只会适得其反，质量越低总体耗时越长。高于这个质量点，想提高质量就得增加成本，质量越高开发时间越长。这样两种观点就统一起来了。

如果在大多数项目中，这个中间质量点是可以作为高质量被接受的，那我们就找到了又快又

好的最佳方法。这个质量点到底是多少？呵，我可以告诉你——87.5%。但是谁知道怎么去度量呢？没有人知道，只能凭感觉和经验了。

## 我们的时间花在哪里

经过这段时间的练习，大多数人都应该体会到了，敲代码并不是耗费时间最多的地方。一个高效率的程序员，并不是说他打字比别人快，而是他节省了别人浪费了的时间。我常说达到别人五倍的效率并不难，因为在软件开发中，大部分人的大部分时间都被浪费掉了，只要把别人浪费掉的时间省下来，你的效率就提高上去了。像在优化软件性能时采用的方法一样，要优化程序员的性能，我们要找出性能的瓶颈。也就是弄清楚我们的时间花在哪些地方，然后想办法省掉某些浪费了的时间。根据我的经验，耗费时间最多的地方包括以下几个方面。

### 分析

需求分析通常是SPEC工程师（或者所谓的系统分析员）的任务。当然，程序员也会参与到这个过程中，但程序员的任务主要是理解需求，然后分析如何实现它们，这个分析工作也就是软件设计。无论是在计算机上用设计工具画出正规的软件架构图，还是在纸上用自然语言描述出算法的逻辑，甚至是在脑海中一闪而过的想法，都可以称之为设计。设计其实就是打草稿，然后反复推敲你的想法，最后得到可行的方案。设计文档只是设计的结果，是设计的表现形式，没有写设计文档，并不代表没有做设计（但是写设计文档可以加深你的思考）。

设计本身是一个思考的过程，需要耗费大量时间，对于新手来说更是如此。理解上一章中那些程序的需求并不难，只需要很少的时间，但大家可能需要花不少时间去思考实现那些程序的方法。这个时间因人而异，有人或许直到最后也没有想出办法，不过这没有关系，没有人天生就会的，不会的原因只是因为你暂时还不知道一些常用的设计方法，甚至大概连基本数据结构和算法都不熟悉。

在后面的章节中，我们会一步步地深入学习各种常用设计方法，反复练习基本数据结构和算法。熟能生巧，软件设计也一样，在你什么都不懂的时候，不可能做出好的设计。你要学习各种经典的设计方法，开始可能只是生搬硬套，多写多练多思考，到后来就可以信手拈来了，从而使设计的时间大大缩短。

### 测试

要写得好自然离不开测试，初学者都有这个概念。他们忠实地使用了教科书上所讲的方法，用`scanf`输入数据，做些操作之后，再用`printf`打印出来。看，这真称得上是一个完美的“输入-处理-输出”过程。测试是要保证正确的输入能产生正确的输出，因此这种方法的原理是没有错的，但使用这种方法的的确确会耗费我们大量时间。

如果测试只需要做一次，这种方法还是可取的，问题是如果每次修改之后都要重复这个过程，那耗费的时间就太多了。这种工作本来就单调乏味，而且很难坚持做下去。要是单元测试做得不全面，就会有更多bug等着去调试。时间久了，或者换人维护了，那么可能谁也搞不清楚什么样的输入会产生什么样的输出，结果到最后可能是连测试也省了（因为没法继续做测试了），那就等着把大量的时间浪费在调试上吧。总而言之，这种测试方法不好，我们需要更有效的测试方法才行。

**调试**

测试时发现程序有bug，自然要用调试器来调试程序了。对一些人来说，调试是一件充满挑战和乐趣的事；而对大部分人来说，特别是对我这种做过两年专职调试的人来说，调试是件无趣又无用的无聊工作。不过对一个程序员而言，熟练使用调试器是必要的，在分析现有软件时，调试器是非常有用的工具。但在开发新软件时，调试器却难免要浪费我们的宝贵时间。

调试器是最后一招，只有在迫不得已时才使用。一位敏捷方法的高手说他已经记不得上次使用调试器是什么时候了，我想这就是敏捷方法能够提高开发速度的原因之所在吧。因为没有什么比一次性写好而不用拿调试器调来调去更快的方法了。

现在大家已经知道了浪费时间的地方，那么在接下来几节中，我将介绍一些避免浪费时间的方法。我希望大家在学完这些方法之后，也能达到普通工程师五倍的效率。最后再啰嗦一句吧，希望大家读完本书所有文章之后，编程水平可以更上一层楼。

## 2.2 代码阅读法

软件工程实践已经证明代码评审是提高代码质量最有效的手段之一，极限编程（XP）更是把代码评审推向极致，形成著名的结对编程工作方式，两个程序员在一台电脑前面工作，一个人编写程序，另一个人评审所输入的每一行代码，写程序的人专注于目前工作的细节，评审的人同时要从更深的层次考虑应该如何改进代码质量，结对的两个人的角色是经常会互换的。

可惜我既没有结对编程的经验，也没有在CMM3（及以上）团队中工作过。不过现在我要介绍比结对编程更敏捷更轻量级，但是同样有效的评审方法。这种方法不需要其他程序员配合，有你自己就够了。为了把这种方法与传统的代码评审区分开来，我把它称为代码阅读法吧。

很多初学者（其实也包括一些有经验的程序员）在敲完代码的最后一个字符后，就会迫不及待地开始编译和运行程序，想马上看到自己的工作成果。这种快速反馈的编程方法有助于满足自己的成就感，但是同时也会带来一些问题。

让编译器帮你检查语法错误确实可以省些时间，但一些程序员往往只关注这些错误，以为改完这些错误就万事大吉了。其实不然，很多错误编译器是发现不了的，如内存错误和线程死锁等，这些错误可能逃过简单的测试而遗留在代码中，直到集成测试或者软件发布之后才暴露出来，那

时再想去修改它们，就要花费很大的代价了。

修改完编译错误之后就是运行程序了，要是运行起来有错误，就轮到调试器闪亮登场了。花了不少时间去调试，到头来却发现自己所犯的无非是些低级错误，或许当时你还会责备自己粗心大意，但是这种一时的自责难保你下次不犯同样的错误。更严重的是，这种“调试加修复”的方法往往只能头痛医头脚痛医脚，治标不治本，最终做出来的还是质量不高的软件。

让编译器帮你检查语法错误，让调试器帮你查bug，这确实是件顺理成章的事，但这确实是又慢又烂的方法。就像你本来是要到离家东边1000米的地方开会，结果你却大步迈向西边，又是坐车又是搭飞机，忙得不亦乐乎，花了不少时间绕着地球转了一周，终于风尘仆仆赶到会议室，你还大发感慨说，现代的交通工具真是发达啊。其实你只要往东走，走路过去也就是十多分钟的事。打这么个看起来很荒诞的比方无非是要说明一个道理——不管你的调试技巧有多高，都不如一次性写好更高效。

我以前也一样，想赶时间，结果却花了更多时间，在经过很多痛苦的经历之后，我开始学会放松自己，有意识地让自己慢下来。在写完程序之后，我会先花些时间去阅读它，一遍、两遍甚至多遍之后，才开始编译它。只要有时间，在通过测试之后，我还会继续阅读这些程序代码。其实每读一遍我都有不同的收获，有时候会发现一些错误，有时候会找到一些可以改进之处，有时候甚至会在脑海里产生全新的想法。

下面是我在阅读自己写的代码时经常用到的一些方法。

#### 检查常见错误

第一遍阅读时主要关注语法错误、代码排版和命名规则等问题，只要看不顺眼就修改它们。读过改过之后，你的代码应该很少有低级错误，而且看起来也比较干净清爽了。第二遍需要重点关注常见的编程错误，比如内存泄露和可能的越界访问、变量没有初始化、函数忘记返回值等问题，在后面的章节中，我还将向大家介绍这些常见错误，避免这些错误可以为你省下大量的时间。如果有时间，在测试完成之后，还可以考虑是否有更好的实现方法可供改进，甚至尝试用一种全新的方法重新去实现这些程序。说了读者可能不相信，在学习编程的前几年，我经常重写整个模块，只因为我觉得这样能让我把程序做得更好，这样能验证我的一些想法或提高我的编程能力，即使连续几天加班到晚上十一点，我也要重写它们。

#### 模拟计算机执行

常见错误是比较死的东西，按照检查列表一条一条照着做就行了。但有些逻辑错误通常就不是这么直观了，这时可以自己模拟计算机去执行，假想你自己是计算机，想象一下你读入这些代码时会怎么处理。这种方法能有效地完善我们的思路。还可以多考虑不同的输入数据和各种边界值，这能帮助我们想到一些没有处理的情况，让程序的逻辑更严谨。

### 假想讲给朋友听

据说在代码评审时发现错误的，往往不是评审的人而是程序员自己。我也有很多这样的经历，在把自己遇到的情况讲给别人听的时候，往往是别人还没有听明白，自己已经发现里面存在的错误了。上大学时，我常常把自己写的或者学到的东西讲给隔壁寝室的一个同学听，他说他从我这里学到了很多知识，其实我从讲的过程中，也经常会发现一些问题，对提高自己的能力大有帮助。可惜并不是随时都能找到好的听众，幸好我们有另外一个替代办法。记得刚开始写程序时看过一本书（忘记名字了），作者说他在写程序时，常常把思路讲给他的布娃娃听。我没有布娃娃当听众，总不至于让我对着鼠标、键盘和显示器讲自己的思路吧，所以我会假想自己身边有个朋友，把自己的思路讲给“他”听，同时也假想“他”来质疑我。话说回来，这种方法确实很有效，能够让自己的思路更清晰，据说一些大师也经常使用这种方法。

这种代码阅读法会花掉你一些时间，但是可以省下更多调试时间，而且能够提高代码质量，可以说是名符其实的“写得又快又好的秘诀”之一。至于读几遍合适，要根据情况而定，我个人觉得读两到三遍是最好的，花费的时间也不算多。

## 2.3 避免常见错误

在C语言中，内存错误是最为人诟病的。这些错误让项目延期或者被取消，引发无数的安全问题，甚至出现人命关天的灾难。抛开这些大道理不谈，它们确实浪费了我们大量时间，这些错误引发的是随机现象，即使有一些先进工具的帮助，为了找到重现的路径，花上几天时间也不足为怪。如果能够在编写代码的时候避免这些错误，开发效率至少提高一倍以上，而且质量也可以提高几倍。这里列举一些常见的内存错误，供新手参考。

### 内存泄露

大家都知道，在堆上分配的内存，如果不再使用了，就应该把它释放掉，以便后面其他地方可以重用。在C/C++中，内存管理器不会帮你自动回收不再使用的内存。如果你忘了释放不再使用的内存，这些内存就不能被重用了，这就造成了内存泄露。

把内存泄露列为首位，倒不是因为它会带来多么严重的后果，而是因为它是最为常见的一类错误。一两处内存泄露通常并不至于让程序崩溃，也不会带来逻辑上的错误，而且在进程退出时，系统会自动释放所有与该进程相关的内存（共享内存除外），所以内存泄露的后果相对来说还是比较温和的。但是，量变会导致质变，一旦内存泄露过多以至于耗尽内存，后续内存分配将会失败，程序就可能因此而崩溃。

现在PC机的内存够大了，再加上进程有独立的内存空间，对于一些小程序来说，内存泄露已经不能构成太大的威胁。但对于大型软件，特别是长时间运行的软件，或者嵌入式系统来说，

内存泄露仍然是致命的。

不管在什么情况下，我们都应该采取谨慎的态度，杜绝内存泄露的出现。相反，认为内存有的是，对内存泄露放任自流都不是负责的。尽管一些工具可以帮助我们检查内存泄露问题，我认为还是应该在编程时就仔细一点，及早排除这类错误，这些工具只应该用来进行验证。

**内存越界访问**

内存越界访问有两种。一种是读越界，即读了不属于自己的数据。如果所读的内存地址是无效的，程序立刻就崩溃了；如果所读内存地址是有效的，在读的时候不会马上出现问题，但由于读到的数据是随机的，因此它会造成不可预料的后果。另外一种是写越界，又叫缓冲区溢出，所写入的数据对别的程序来说是随机的，它也会造成不可预料的后果。

内存越界访问会造成非常严重的后果，是程序稳定性的致命威胁之一。更麻烦的是，它出现的时机是随机的、表现出来的症状是随机的，而且造成的后果也是随机的。这会使程序员很难找出这些bug的现象和本质之间的联系，从而给bug的定位带来极大的困难。

虽然一些工具可以帮助你检查内存越界访问的问题，但也不能过于依赖工具。内存越界访问通常是动态出现的，往往只在极端的情况下才会出现，除非精心设计测试数据，否则工具也无能为力。而且工具本身也有一些限制，甚至在一些大型项目中，工具变得完全不可用。比较保险的方法还是在编程时就多加小心，特别是要仔细检查外部传入的参数。

我们来看一个例子。

```
#include <stdlib.h>
#include <string.h>

int main(int argc, char* argv[])
{
        char str[10];
        int array[10] = {0,1,2,3,4,5,6,7,8,9};

        int data = array[10];
        array[10] = data;

        if(argc == 2)
        {
            strcpy(str, argv[1]);
        }
        return 0;
}
```

这个例子中有两个错误是新手常犯的。

其一：int array[10]定义了10个元素大小的数组，由于C语言中数组的索引是从0开始的，

所以只能访问array[0]到array[9]，访问array[10]就造成了越界错误。

其二：strcpy(str, argv[1]);这里是否存在越界错误依赖于外部输入的数据，这样的写法在接受正常输入的情况下可能没有问题，但只要受到一点恶意攻击就完蛋了。除非你确定输入数据是在你的控制之内的，否则不要用strcpy、strcat和sprintf之类的函数，而要用strncpy、strncat和snprintf代替。



**野指针**

野指针说的是那些指向你已经释放掉的内存的指针。当你调用free(p)时，你真正清楚这个动作背后的内容吗？你会说p指向的内存被释放了。没错，那么p本身有变化吗？答案是p本身没有发生任何变化。它指向的内存仍然是有效的，就算你要继续读写p指向的内存，也没有人能拦得住你。

释放掉的内存会被内存管理器重新分配，此时，野指针指向的内存就已经被赋予新的意义。对野指针指向内存的访问，无论是有意还是无意的，都可能会带来巨大的代价，因为它造成的后果，如同越界访问一样是不可预料的。

释放内存后应当立即把对应指针置为空值，这是避免野指针常用的方法。这个方法简单有效，只是要注意，如果指针是从函数外层传入的，那么就算在函数内把指针置为空值，也不会对外层的指针造成任何影响。比如，在析构函数里把this指针置为空值是没有任何效果的，这时应该在函数外层把指针置为空值。

**访问空指针**

空指针在C/C++中占有特殊的地址，通常它是用来判断一个指针的有效性的。空指针一般定义为0。现代操作系统都会保留从0开始的一块内存，至于这块内存有多大，视不同的操作系统而定。一旦程序试图访问这块内存，系统就会触发一个异常/信号。

操作系统为什么要保留一块内存，而不是仅仅保留一个字节的内存呢？这是因为一般内存管理都是按页进行管理的，根本就无法单纯保留一个字节，而至少要保留一个页面。保留一块内存也有额外的好处，可以检查诸如p=NULL; p[1]之类的内存错误。

在一些嵌入式系统中（如ARM7），从0开始的一块内存是用来安放中断向量的，没有MMU的保护，直接访问这块内存好像并不会引发异常。不过这块内存是代码段的，而不是程序中有效的变量地址，所以用空指针来判断指针的有效性仍然可行。

**引用未初始化的变量**

未初始化变量的内容是随机的（有的编译器会在调试版本中把它们初始化为固定值，如0xcc），使用这些数据会造成不可预料的后果，调试这样的bug也是非常困难的。

对于态度严谨的程度员来说，要防止这类bug出现其实非常容易，只需要在声明变量时记得对它进行初始化就行，这是一个好的编程习惯，希望大家能够在学习编程中逐渐养成这个习惯。另外我们也要重视编译器的警告信息，一旦发现引用了未初始化的变量，就要立即修改过来。

在下面这个例子中，全局变量g_count是确定的，因为它在bss段中，已经是自动初始化为0了。临时变量a是没有初始化的，堆内存str也是没有初始化的。但这个例子有点特殊，由于程序刚运行起来，很多东西都是确定的，因此如果你想把它们当作随机数的种子是不行的，因为它们还不够随机。

```
#include <stdlib.h>
#include <string.h>

int g_count;

int main(int argc, char* argv[])
{
        int a;
        char* str = (char*)malloc(100);

        return 0;
}
```

**不清楚指针运算**

对于一些新手来说，指针常常让他们犯糊涂。

比如，int *p = ...; p+1等于(size_t)p + 1吗？

老手们自然心里清楚，但新手可能就弄不太清了。

事实上，p+n 等于 (size_t)p + n * sizeof(*p)。

指针是C/C++中最有力的武器，功能非常强大，无论是变量指针还是函数指针，都应该掌握得非常熟练。只要有不确定的地方，就应该马上写个小程序验证一下。如果对每一个细节都了然于胸，可以在编程时省下不少时间。

**结构的成员顺序变化引发的错误**

在初始化一个结构时，老手可能很少像新手那样老老实实地、一个成员一个成员地为结构初始化，而是采用“走捷径”的方式。

```
Struct s
{
        int   l;
        char* p;
};
```

```
int main(int argc, char* argv[])
{
        struct s s1 = {4, "abcd"};

        return 0;
}
```

使用以上这种方式是非常危险的行为，原因在于你这样做就相当于对结构的内存布局作了假设。如果这个结构是第三方提供的，他很可能调整了结构中成员的相对位置。而这样的调整往往不会在文档中说明，你自然很少去关注。如果调整的两个成员具有相同数据类型，编译时不会有任何警告，而程序的逻辑可能相差十万八千里了。

正确的初始化方法应该是（当然，一个成员一个成员地初始化也行）。

```
struct s
{
        int   l;
        char* p;
};

int main(int argc, char* argv[])
{
        struct s s1 = {.l=4, .p = "abcd"};

        return 0;
}
```

**注意** 有的编译器可能不支持新标准。

#### 结构的大小变化引发的错误

我们看看下面这个例子。

```
struct base
{
        int n;
};

struct s
{
        struct base b;
        int m;
};
```

在面向对象编程中，我们可以认为这是第二个结构继承了第一结构，有什么问题吗？当然没有，这是C语言中实现继承的基本手法。

现在假设第一个结构是第三方提供的，第二个结构是你自己的。第三方提供的库是以DLL方式分发的，DLL最大好处在于可以独立替换。但随着软件的进化，问题可能就来了。

假设第三方在第一个结构中增加了一个新的成员int k;，编译好后把DLL给你，你不管不问就直接把它给了客户，让他们替换掉老版本。这样的话，程序加载时倒不会有任何问题，但是它的运行逻辑可能完全改变！原因是两个结构的内存布局重叠了。

解决这类错误的唯一办法就是重新编译全部代码。由此看来，动态库并不见得可以动态替换，如果你想了解更多相关内容，建议你阅读《COM本质论》。[①]

**分配/释放不配对**

大家都知道malloc要和free配对使用，new要和delete/delete[]配对使用，重载了类的new操作，应该同时重载类的delete/delete[]操作。这些都是千千万万本书上反复强调过的，除非当时晕了头，否则一般不会也绝对不应该犯这样的低级错误。

而有时候我们却被蒙在鼓里，可能有两段代码看起来都是调用的free函数，实际上却调用了不同的实现。比如在Win32下，调试版与发布版、单线程与多线程使用的是不同的运行时库，而不同的运行时库使用的是不同的内存管理器。如果一不小心链接错了库，那你就麻烦大了。程序可能很容易崩溃，原因很简单：在一个内存管理器中分配的内存，在另外一个内存管理器中释放时就会出现问题。

**返回指向临时变量的指针**

大家都知道，栈里面的变量都是临时的。当前函数执行完成时，相关的临时变量和参数就都被清除了。不能把指向这些临时变量的指针返回给调用者，因为这样的指针指向的数据是随机的，会给程序造成不可预料的后果。

下面是个错误的例子。

```
char* get_str(void)
{
        char str[] = {"abcd"};

        return str;
}

int main(int argc, char* argv[])
{
        char* p = get_str();

        printf("%s\n", p);
```

① 已由中国电力出版社出版。——编者注

```
        return 0;
}
```

下面这个例子则没有问题，大家知道为什么吗？

```
char* get_str(void)
{
        char* str = {"abcd"};

        return str;
}

int main(int argc, char* argv[])
{
        char* p = get_str();

        printf("%s\n", p);

        return 0;
}
```

### 试图修改常量

在函数参数前加上const修饰符，是给编译器做类型检查用的，编译器会禁止修改这样的变量。但这并不是强制的，你完全可以用强制类型转换绕过去，一般也不会出什么错。

而对全局常量和字符串而言，就算用强制类型转换绕过去，运行时仍然会出错。原因在于它们是放在.rodata里面的，而.rodata内存页面是不能修改的。试图对它们进行修改是会引发内存错误的。

下面这个程序在运行时会出错。

```
int main(int argc, char* argv[])
{
        char* p = "abcd";
        *p = '1';

        return 0;
}
```

### 误解传值与传引用

在C/C++中，参数默认传递方式是传值的，即在参数入栈时被复制一份。在函数里修改这些参数，不会影响外面的调用者。

```
#include <stdlib.h>
#include <stdio.h>
```

```
void get_str(char* p)
{
        p = malloc(sizeof("abcd"));

        strcpy(p, "abcd");

        return;
}

int main(int argc, char* argv[])
{
        char* p = NULL;

        get_str(p);

        printf("p=%p\n", p);

        return 0;
}
```

在main函数里，p的值仍然是空值。当然在函数里修改指针指向的内容是可以的。

### 符号重名

无论是函数名还是变量名，如果在不同的作用范围内重名，自然没有问题。但如果两个符号的作用域有交集，如全局变量和局部变量，全局变量与全局变量之间，重名的现象一定要坚决避免。gcc有一些隐式规则来决定处理同名变量的方式，编译时可能没有任何警告和错误，但结果通常并非你所期望的。

下面例子中，t.c和main.c两段代码在编译时就没有警告。

```
/*t.c*/
#include <stdlib.h>
#include <stdio.h>

int count = 0;

int get_count(void)
{
        return count;
}

/*main.c*/
#include <stdio.h>

extern int get_count(void);

int count;

int main(int argc, char* argv[])
{
        count = 10;
```

```
        printf("get_count=%d\n", get_count());

        return 0;
}
```

但如果把main.c中的int count;修改为int count = 0;，在编译时gcc就会报错，说multiple definition of 'count'（count有多个定义）。它的隐式规则比较奇妙吧，所以还是不要依赖它为好。



**栈溢出**

我们在前面关于堆和栈的一节讲过，在PC上，普通线程的栈空间也有十几MB，通常够用了，定义大一点的临时变量一般不会有什么问题。

而在一些嵌入式系统中，线程的栈空间可能只有5KB大小，甚至小到只有256个字节。在这样的平台中，栈溢出是最常见的错误之一。

在编程时自己应该清楚平台的限制，从而能采取必要措施避免栈溢出的可能。

**误用sizeof**

尽管C/C++通常是按值传递参数，但数组是个例外，在传递数组参数时，数组退化为指针（即按引用传递），用sizeof是无法取得数组的大小的。

从下面这个例子可以看出。

```
void test(char str[20])
{
        printf("%s:size=%d\n", __func__, sizeof(str));
}

int main(int argc, char* argv[])
{
        char str[20]  = {0};

        test(str);

        printf("%s:size=%d\n", __func__, sizeof(str));

        return 0;
}
```

上面的程序会输出以下结果。

```
[root@localhost mm]# ./t.exe
test:size=4
main:size=20
```

### 字节对齐

字节对齐主要目的是提高内存访问的效率。但在有的平台（如ARM7）上，就不光是效率问题了，如果不对齐，得到的数据是错误的。

所幸的是，大多数情况下，编译器会保证全局变量和临时变量能按正确的方式对齐。内存管理器会保证动态内存按正确的方式对齐。要注意的是，在不同类型的变量之间进行转换时一定要小心，如把char* 强制转换为int* 时，就要格外地小心。

另外，字节对齐也会造成结构大小的变化，如果是在程序内部，用sizeof来取得结构的大小就足够了。若数据要在不同的机器间传递时，就需要在通信协议中规定对齐的方式，以避免对齐方式不一致引发的问题。

### 字节顺序

字节顺序历来是设计跨平台软件时头疼的问题。字节顺序是关于数据在物理内存中的布局的问题，最常见的字节顺序有两种：大端模式与小端模式。

大端模式是高位字节数据存放在内存低地址处，低位字节数据存放在内存高地址处。
小端模式指低位字节数据存放在内存低地址处，高位字节数据存放在内存高地址处。

在普通软件中，字节顺序问题并不引人注目。而在开发与网络通信和数据交换有关的软件时，就要特别注意字节顺序问题了。

### 多线程共享变量没有用valotile修饰

关键字valotile的作用是告诉编译器，不要把变量优化到寄存器里。在开发多线程并发的软件时，如果这些线程共享一些全局变量，这些全局变量最好用valotile修饰。这样可以避免因为编译器优化而引起的错误（这样的错误非常难查）。

### 忘记函数的返回值

函数需要返回值。如果你忘记return语句，它仍然会返回一个值，因为在i386及更高级的微处理器中，有EAX寄存器可以用来保存返回值，如果没有明确返回一个值，就会返回EAX中最后保存的内容。

## 2.4 自动测试

虽然比不做测试多少要好一些，但是手工测试却存在不少的问题：手工输入数据的过程单调乏味，很难长期坚持；每次都要重新输入数据，浪费大量时间；测试用例不能累积，测试往往不

完整；用人脑判断输出的正误，浪费人力也存在误差。这些问题都使得手工测试在实际应用中有着很大的局限。要写得好，测试自然不能省；要写得快，那就需要更好的测试方法出马了。

更好的测试方法当然是自动测试了。幸运的是，刚进入这个行业我就接触了自动的测试（呵，读本文的初学者就更幸运了），我的第一份正式工作是在测试组写测试程序。当时测试组也算是人才济济了，还有几个北大毕业的，不过他们都不懂Linux，所以上司指派我去为移植到Linux上的模块写测试程序。这些模块其实都有测试程序，但这些测试程序的功能太弱了，我的上司也曾要求开发人员改进测试程序，但那些开发人员根本不以为意，完全不理会我们的要求，所以我们只好自己重写这些测试程序。总共有50多个模块，熟悉这些模块需要不少时间，按每两个工作日写一个测试程序来算，上司给了我5个月时间。

记得第一个模块是RDFParser。RDF（Resource Description Framework，资源描述框架）是XML的一种应用，RDFParser实际上就是一个XML解析器，而且它被包装成符合RDF要求的接口。当时我对C/C++还不太熟悉，对RDF更是陌生，因此我花了两周时间才写出这个测试程序。程序运行起来有些不正常，但我确信这不是测试程序本身的问题，就去请开发人员帮忙来看一下。负责RDFParser的那个程序员是北京某著名高校毕业的，不过我真是没有见过第二个比他更自以为是的程序员了——他刚在我座位上坐下就很大声地说：“你们QA的人太蠢了！”

听他这么一说，我当场就愣住了，不过因为初到公司，见上司都没反应，我自然也就忍了。我列举了一些证据，说明这应该是模块里面的问题，他根本就不听，只是不断重复地表示这不可能是他程序的问题，而是我们QA的人太蠢了，总是在浪费他的时间。过了一会儿，他终于闭上了嘴巴，又过了一会儿，他才对我说：“等会儿重新发个版本给你吧。”后来又请他过来四五次，结果每次都是他的问题。

之后我再没有听到他说过“你们QA的人太蠢了”这样的话。为了避免让他抓到把柄来嘲笑测试组，我决定在请他来查问题之前做更详细的测试。当时我写的测试程序和现在初学者写的测试程序没有两样，都是从教科书上学来的，先通过`scanf`从终端输入数据，调用被测函数，再把结果`printf`出来，这花了我太多时间。一想到后面还有50多个模块的测试程序要写，我知道这样下去肯定不行，一定得想个更好的办法。

后来我把输入的数据和期望的结果都写到一个INI文件中，测试程序从这个文件中读入数据，运行测试，再将输入与预期结果相比较，整个过程都自动化了。写INI文件的解析器花了我一周时间，后来又重写了那个测试程序，结果是完成RDFParser的测试程序花了我整整一个月的时间。进度上自然是大大落后了，还好上司知道事情的来龙去脉后并没有责备我，只是让我慢慢做就好了。

写第二个测试程序时把INI解析器的代码复制过去，再加一些调用模块的代码就算写好了，第三个也是如此。写了几个之后，我发现了INI解析器有个bug，结果就是我不得不去修改每个测

试程序，我意识到之前那样做会让维护工作很麻烦，于是把INI解析器的接口规范化了，并编译成一个独立共享库。之后，我又写了几个测试程序，写着写着就觉得不耐烦了，因为这些测试程序无非都是读入数据，调用被测函数，再检查结果，这个过程就是个很机械化的过程，实在是太无聊了。想到后面还要把这个机械的过程再重复个几十遍，我很是郁闷。不过在郁闷了几天之后，我突然灵机一动，决定写一个代码生成器来自动生成这些代码。开始的代码生成器是用C写的，用一个简单的规则来描述被测函数，通过这些规则来产生测试程序。我把这些东西和INI解析器放在一个独立的库中，就成了一个TesterFrameWork，经过几个测试程序的验证和完善，后来利用这个TesterFrameWork，只要一两个小时就能完成一个测试程序了。有次请开发人员中的一个高手帮我查一个问题，他看了一会儿我的TesterFrameWork之后，盯着我说："你太聪明了。"我笑了笑告诉他："其实我才刚刚开始写C/C++程序。"

一年之后，我知道了有个叫做CPPUnit的工具，为了赶时髦，我把TesterFrameWork改名为CxxUnit，非典爆发那年的时候，因为放假无聊我就把它重写了一遍放在cosoft上了（之后没有管过它，或许还在吧）。

对整个大系统进行自动测试是很难的，独立的模块才是最佳的自动测试单元。这样一来，自动测试和单元测试几乎成了等价的概念。很多人可能会认为自动测试就是利用CPPUnit这样的单元测试框架随意写个测试程序而已，这种观点实在是很荒谬，你总不能认为有了个好的设计文档的模板，就能照着填空填出一个好设计吧。

我自己曾亲自实现过单元测试框架，这可不是像有些人那样出于模仿去实现，而是完全出于实际的需要，后来我也研究其他测试框架，因此，我敢说我对测试程序框架的认识比一般程序员要深刻。我认为测试程序框架可以减化一些测试程序的工作，但它与自动测试之间关系并不密切，用不用测试程序框架完全是个人喜好。用测试程序框架未必能写出好的测试程序，就像用了C++也未必能写出好的面向对象的程序一样。

虽然顺利地完成了那个写测试程序的任务，但我一直被一些问题困扰。我该如何写测试用例？我该如何去检测结果？这些工作都是测试程序框架帮不上忙的。写测试用例还好说，通过边界值法、等价类法和路径覆盖法就可以找到最常用的测试用例。检测结果呢？有人说很简单啊，判断返回值就好了。那我问一下`dlist_insert`返回OK，就真的OK了吗？如果一个函数根本没有返回值，那你怎么判断呢？

测试程序框架是敏捷论者所提倡的，但在我看来，它本身根本不够敏捷：你不仅要去学习它，了解它的运行机制，还要在使用它时包含它的头文件，链接它的库。就没有比它更敏捷的方法了么？重要的是，它可能根本帮不上什么忙。前面的问题折磨了我一段时间，我于是得出一个可能有点偏激的结论：测试程序框架都是愚蠢的，它根本没法为你提供你真正需要的帮助（我知道这样说会得罪一些乐于使用测试程序框架的朋友，如果你真想找我讨论这个问题的话，请看完本节的附带示例代码再说）。

就在那个时候，我有幸读到了孟岩老师翻译的《Design by Contract原则与实践》[①]，读完之后我豁然开朗。或许我还没有明白契约式设计的本质，但我确实知道了写自动测试程序的方法，下面我介绍一下。

- 在设计时，每个函数只完成单一的功能。单一功能的函数容易理解，也容易预测其行为。对测试来说，给定一些输入数据，就能知道它的输出和影响，这样的函数是最容易测试的。
- 在设计时，把函数分为查询和命令两类。查询函数只查询对象的状态，而不改变对象的状态。命令函数则只修改对象的状态，并返回其操作是否成功的标志，而不会返回对象的状态。比如，dlist_length查询双向链表的长度，它不修改双向链表的任何状态。dlist_delete修改对象的状态（删除结点），并返回其操作是否成功，而不返回当前长度或者结点是否存在之类的状态。
- 在设计时，把查询分为基本查询和复合查询两类。基本查询函数只查询单一的状态，而复合查询可以同时查询多个状态。比如，window_get_width返回窗口的宽度，这是基本查询函数，widget_get_rect返回窗口的左上角坐标、宽度和高度，这是复合查询函数。
- 在实现时，检验输入数据，确认使用者正确地调用了函数。契约式设计规定了调用者和实现者双方的责任：调用者需要使用正确的参数，以保证有正确的结果；而实现者则需要检查输入参数是否违背了契约。

那应该怎样检查输入参数呢？有人或许会说，检查到无效参数就返回一个错误码。这当然可以，只是不太好，因为大多数人都没有检查返回值的习惯，而且在每个地方都检查函数的返回值是件很繁琐的事，也会让代码看起来比较乱。通常我们只检查一些关键的地方，这样一来，一些无效参数这样的错误可能就无声无息地隐藏起来了，这样实在很糟糕，因为错误隐藏得越深，发现它的时间就越晚，修改它的代价就越大。

在C++和Java里，如果参数不正确，通常是抛出一个无效参数之类的异常，C语言里面没有异常这个概念，我们需要其他办法才行。有人推荐用assert来检查，这是一个好办法，assert只在调试版本中有效（没有定义NDEBUG），这样任何无效调用都在调试版本中暴露出来了。如果配合前面返回错误码的方法，在发布版本中应该能避免程序毫无征兆地死掉。使用方法如下。

```
assert(thiz != NULL);
if(thiz == NULL)
{
    return DLIST_RET_INVALID_PARAMS;
}
```

我一直使用这种方法，但是有个问题——这样做无法用自动测试验证assert是否正常地触

① 已由人民邮电出版社出版。——编者注

发了。当用错误的参数测试时，我期望assert被触发，但如果assert被触发了，自动程序测试就死掉了，一旦自动测试程序死掉，就无法继续验证下一个assert了。这是一个悖论！

后来我从glib里面学了一招，在检查时不用assert，而只是打印出一个警告，代码也很简明，按它的方式，我们这样检查。

```
return_val_if_fail(cursor != NULL, DLIST_RET_INVALID_PARAMS);
```

我们需要定义两个宏，一个用于无返回值的函数，一个用于有返回值的函数。

```
#define return_if_fail(p) if(!(p))                              \
    {printf("%s:%d Warning: "#p" failed.\n",                     \
        __func__, __LINE__); return;}
#define return_val_if_fail(p, ret) if(!(p))                     \
    {printf("%s:%d Warning: "#p" failed.\n",                     \
__func__, __LINE__); return (ret);}
```

这样一来，遇到无效参数时，我们可以看到一个警告信息，同时又不会影响自动测试。

- 在测试时，用查询来验证命令。命令一般都有返回值，但只检查返回值是不够的。比如，dlist_delete返回了OK，它就真的OK了吗？我们信任它，但还是要检查。那么应该怎么检查？

  很简单，用查询函数来检查对象的状态是不是我们所希望的。

  对于dlist_delete，我们做出如下预期。

  (1) 如果输入无效参数，则期望返回DLIST_RET_INVALID_PARAMS。

  (2) 如果输入正确参数，则期望：函数返回DLIST_RET_OK；双向链表的长度减1；删除的位置的下一个元素被移到删除的位置。

  在测试程序中检查时，因为任何不符合期望的结果都是bug，所以我们用assert检查。这样有问题马上就暴露出来了，定位错误比较容易，通常都不需要调试器。我们这样来检查。

```
assert(dlist_length(dlist) == (n-i));
assert(dlist_delete(dlist, 0) == DLIST_RET_OK);
assert(dlist_length(dlist) == (n-i-1));
if((i + 1) < n)
{
    assert(dlist_get_by_index(dlist, 0, (void**)&data) == DLIST_RET_OK);
    assert((int)data == (i+1));
}
```

  （完整的例子请看本节的示例代码。）

- 在测试时，用基本查询去验证复合查询。基本查询和复合查询返回的结果应该一致。比如：

```
Rect rect = {0};
    widget_get_rect(widget, &rect);
    assert(widget_get_width(widget) == rect.width);
    assert(widget_get_height(widget)== rect.height);
```

- 在测试时，预期结果依赖其执行上下文，我们要按逻辑组织测试用例。前面调用的函数可能改变了对象的状态，为了简化测试，在每组测试用例开始时，都重置对象到初始状态。
- 在测试时，第一次只写基本的测试用例，以后逐渐累积，每次发现新的bug就把相应的测试用例加进去。每次修改了代码就运行一遍自动测试，保证修改没有引起其他副作用。

按着上面的原则，应付正常模块的测试没有问题了，但是下面三种情况仍会让你觉得比较棘手。



(1) 带有GUI的应用程序。有GUI的程序会给自动的数据输入和结果检查带来麻烦，有些工具可以部分解决这个问题，特别是针对Win32下的GUI，因为我很少在 Windows下写程序，所以对这方面了解不多。不过最好的办法还是用MVC模型等方式分离界面和实现，因为界面通常相对比较简单，可以手工测试，而实现的逻辑比较复杂，这部分可以自动测试。后面我们会专门讲解分离界面和实现的方法。

(2) 有随机数据输入。如果有些输入数据是内部随机产生的（比如游戏中随机的步骤和无线网络信号的变化等），那你根本无法预测它的输出结果和影响。对于我们可以控制的随机数据，可以提供额外的函数去获取这些数据。对于无法控制的随机输入数据，可以把它们隔离开，即在自动测试中使用固定的数据。

(3) 多线程运行的程序。多线程的程序也很难自动测试，比如向链表中插入一个元素，当你检查的时候，根本不能确定链表的长度是否增加，也无法知道处于刚才插入位置的元素是否是你插入的元素，因为这个时候，可能有另外一个线程已经把它删除了或者加入了新的数据。不过在单线程的自动测试通过之后，多线程的问题会大大减少，剩下的问题我们可以通过其他方式加以避免。

写自动测试程序会花费一些时间，但这项投资能带来最大的回报：减少后面调试时的浪费，提高代码的质量，更重要的是，你终于可以睡个安稳觉了。

## 2.5 Save your work

> “Ernst和Young所在的小组决定使用正规的开发理论——他们常用削减法，分阶段进行开发并具有中途交付能力。他们的步骤包括细致的分析和设计——正如本章描写的基本原则一样。而其他竞争者径直开始了编码，在开始几个小时里，Ernst和Young小组落后了。但到中午时，Ernst和Young小组却遥遥领先了，而到了这一天结束的时候，他们却失败了。导致失败的原因不是因为他们使用了正规方法，而是他们偶然出错把工作文件覆盖了，最终他们比午餐时所做的估计少交付了一些功能，他们败在了没有使用有效的源程序版本控制这个典型的错误上。”
>
> ——摘自《快速软件开发》[①]

① 已由清华大学出版社出版。——编者注

前段时间看探索频道的“荒野求生秘技”（Man & Wild），我很喜欢这个节目也喜欢节目里的那个英国人，甚至连重播都不会放过。他在这个节目里展示了在沙漠、丛林、冰河和雪山等各种环境的求生秘技，他吃过蜘蛛、白蚁、蝎子和蜥蜴，边吃还边说这东西很恶心，但是里面含有非常丰富的维生素、蛋白质和糖分，能够“Save your life”，所以就算恶心也要吃下去。

在Man & Code的世界里，环境好多了，不用面临危险，而且寻找水源和食物根本不需要什么秘诀。这里我们不需要求生秘技去Save your life，但我们需要一些习惯去Save your work。我说过要成为一名高效的程序员，不是因为他打字比别人快，而是因为他省下了别人浪费的时间。说到浪费时间，有什么比成果被毁、从头再来更浪费时间呢？下面我将向大家介绍一些习惯，它们简单有效，根本算不上什么秘技，但它们确实能够Save your work，让你的工作稳步前进。

### 随时存盘

每次停电时，我都会听到有人惊呼：“完了，我的代码没有保存！”补回半小时或一个小时的工作不难，在一个好的工作环境里，这种情况一年也就会遇到那么寥寥几次，浪费的时间完全可以忽略不计。但是那种失去自己工作成果的感觉真的让人很难受，会影响你的工作情绪，无缘无故地让你重做你的工作，和因为要改进去重做完全是两回事。在我以前工作过的一个公司，有段时间经常跳闸，每周都要停电好几次，但是怎么也找不到原因，后来请人来查，据说是线路太长，静电引起的跳闸。在经过那段时间的折磨后，我养成了一个习惯：写代码的时候，平均每30秒钟存盘一次。现在再遇到停电的情况，在别人惊呼的时候，我却可以开始闭目养神了。

### 使用版本控制系统

和一些老程序员聊天时（呵，其实我也老了），他们经常会问我们的项目有没有使用版本控制系统。我说当然有，从大二的时候开始，我就会用Sourcesafe来管理用PowerBuilder写的代码了，后来参加工作了也是一直在使用不同的版本控制系统。接着他们就开始向我“哭诉”他们惨痛的经历……这些经历小则让他们项目延期，大则导致整个项目失败。

版本控制系统有很多功能，但对我个人来说，它最重要的功能是备份代码的。每完成一个小功能，我都会把它提交（checkin）进去，如果我不小心删除了本地文件，或者某个做尝试的修改失败了，我可以把代码恢复到前一个版本。不同团队有不同的规则，有的团队是不允许按这种方式提交代码的，他们只允许提交经过严格测试的代码。如果是那样，你可以在本地建立自己的版本控制系统，初学者在学习时也可以这样做。现在有很多免费的版本控制系统可用，像CVS、SVN和GIT等等，我个人习惯用CVS，SVN是CVS的改进版，将来肯定会替代CVS的，所以推荐大家使用它。

**定期备份**

温伯格在《质量软件管理：系统思维》[①]一书中讲了一个有趣的故事，他以前去研究一些失败的案例，发现这些项目的失败都是因为运气欠佳导致的——比如遭受洪水、地震、火灾和流行感冒等灾害的影响，项目主管们把自己描述成外部问题的受害者。他又对另外一些成功的项目进行研究，发现其中有些项目同样经历了这些自然灾害，但是他们成功地完成了任务。区别只是在于成功项目的主管往往会采用积极预防措施，如定期备份代码并把它们放到不同的地点。



以前在学校的时候，我有两台电脑，一台赛扬和一台486。我经常在上面重装系统，一会儿装Linux，一会儿装NT，一会儿又装NetWare。虽然我经常会把代码备份到不同的分区上以防不测，但我真不缺少因为不小心而把所有分区全干掉的惨痛经历。好在那时我写的只是一些小程序，即使重写一遍问题也不大。但是对于专业程序员或一个软件团队来说，重写所有代码是不可接受的，所以需要更可靠的备份机制。

使用源代码管理系统还不足以保证代码的安全，比如服务器硬盘损坏和办公室发生火灾等情况都是有可能发生的。因此，团队里一定要有人负责定期备份源代码管理系统上的资料，作为初学者也应该有这种意识。另外，我发现有些朋友会选择把重要的资料放在邮箱里，现在的邮箱容量很大，邮箱服务提供商还会定期备份用户的数据，非常安全，而且只要有网络连接的地方就能访问这些数据，非常方便，所以把重要资料放在邮箱里倒也是一个不错的主意。

**状态不好就做点别的**

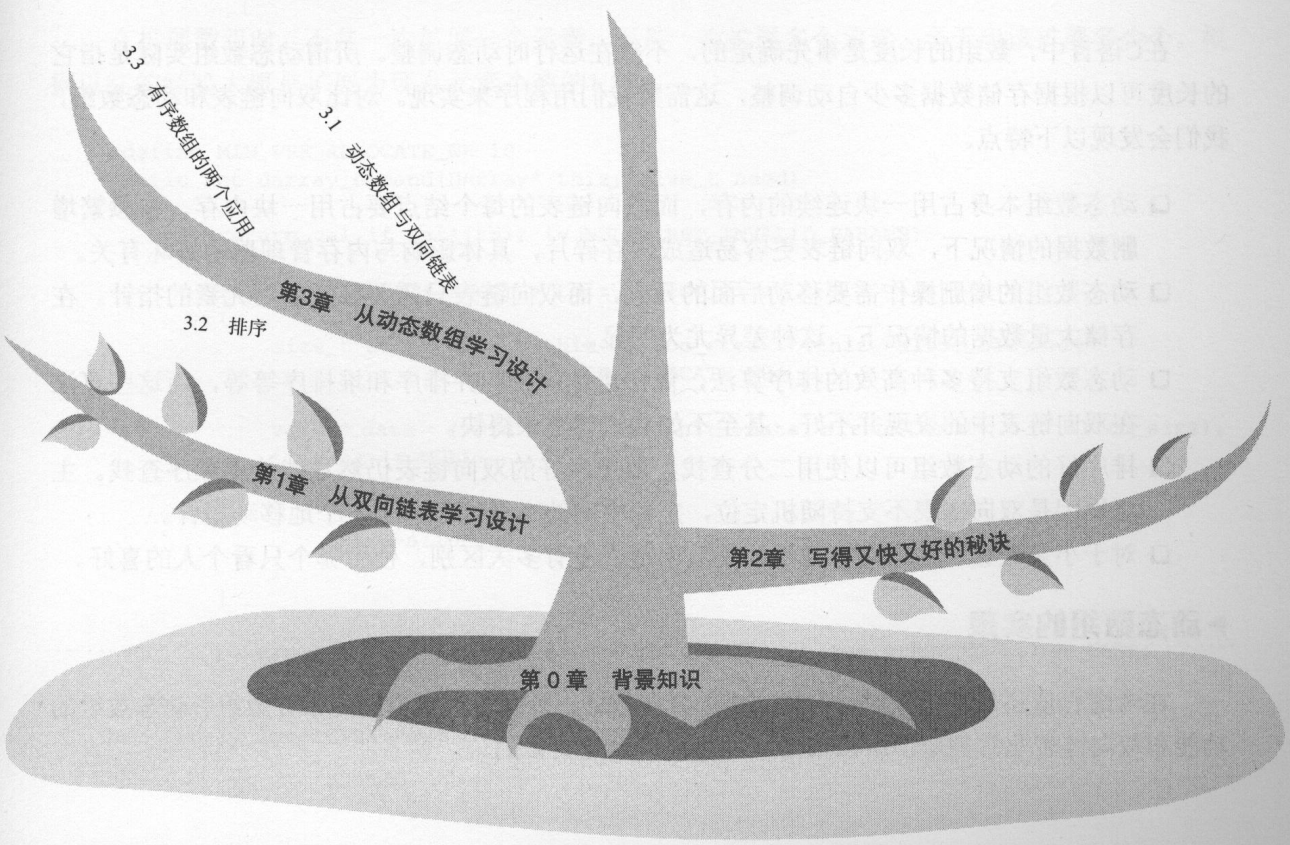
女同胞有定期状态不佳的时候，男同胞也不是每天状态都很好。感冒了、丢东西了，或者是和家人争吵了，都会影响你的状态。状态不好的时候做事，往往是进一步退两步，甚至犯下严重的错误。有次我得了重感冒，居然在服务器的根目录下运行`rm * -rf`（删除全部文件），由于删除过程时间很长，我不久就发现删错地方了，结果吓得我出了一身冷汗，还好那台服务器没有运行着源代码管理系统，但最后还是浪费了我两天时间去重建服务器上的环境。

状态不好的时候，编程也会犯一些低级错误，这让你需要花费更多时间去调试。总而言之，状态不好的时候还要“坚持”完成关键工作是有百弊而无一利，这时你不妨去做点别做的事情，比如看看其他模块的代码之类的，就算你完全放松去休息，也远比犯下严重的错误强。

① 已由清华大学出版社出版。——编者注

# 第 3 章

# 从动态数组学习设计



## 3.1 动态数组与双向链表

双向链表和动态数组是最简单的两种数据结构，在研读大量源代码之后，我发现正是这两种最简单的数据结构，却有着最广泛的应用。像平衡二叉树这样的复杂数据结构，你或许不会有机会在开发中用到它，甚至可能除了在学数据结构时会练习一下之外，永远都不会写第二遍，而双向链表和动态数组则会在开发中反复地运用。也正是由于这个原因，我才选择双向链表和动态数组作为学习的载体。

这里请读者做两件事。

(1) 思考双向链表和动态数组的特点及适用条件。

虽然学习了基本的数据结构和设计方法，但很多人在解决实际问题时还是束手无策：他们不知道什么时候应该用什么数据结构和设计方法。所以在学习的过程中，还应该经常问这个问题——这个东西在什么时候能派上用场？千万不要为了学习而学习，而是要为实际应用而学习。

(2) 按前面在双向链表中学习到的方法实现动态数组。

### ▶双向链表和动态数组的对比

在C语言中，数组的长度是事先确定的，不能在运行时动态调整。所谓动态数组实际是指它的长度可以根据存储数据多少自动调整，这需要我们用程序来实现。对比双向链表和动态数组，我们会发现以下特点。

- ❑ 动态数组本身占用一块连续的内存，而双向链表的每个结点要占用一块内存。在频繁增删数据的情况下，双向链表更容易造成内存碎片，具体影响与内存管理器的好坏有关。
- ❑ 动态数组的增删操作需要移动后面的元素，而双向链表只需要修改前后元素的指针。在存储大量数据的情况下，这种差异尤为明显。
- ❑ 动态数组支持多种高效的排序算法，像快速排序、归并排序和堆排序等等，而这些算法在双向链表中的表现并不好，甚至不如冒泡排序来得快。
- ❑ 排序好的动态数组可以使用二分查找，而排序好的双向链表仍然只能使用顺序查找。主要原因是双向链表不支持随机定位，定位中间结点只能靠一个一个地移动指针。
- ❑ 对于小量数据，使用动态数组还是双向链表没有多大区别，使用哪个只看个人的喜好。

### ▶动态数组的实现

在考虑存值还是存指针时，我们同样选择存指针，所以这里实现的是指针数组。动态数组的功能和双向链表非常类似，所以二者对外的接口也是类似的。

```
struct _DArray;
typedef struct _DArray DArray;

DArray* darray_create(DataDestroyFunc data_destroy, void* ctx);

Ret    darray_insert(DArray* thiz, size_t index, void* data);
Ret    darray_prepend(DArray* thiz, void* data);
Ret    darray_append(DArray* thiz, void* data);
Ret    darray_delete(DArray* thiz, size_t index);
Ret    darray_get_by_index(DArray* thiz, size_t index, void** data);
Ret    darray_set_by_index(DArray* thiz, size_t index, void* data);
size_t  darray_length(DArray* thiz);
int    darray_find(DArray* thiz, DataCompareFunc cmp, void* ctx);
Ret    darray_foreach(DArray* thiz, DataVisitFunc visit, void* ctx);

void darray_destroy(DArray* thiz);
```



动态数组的动态性是如何实现的呢？其实很简单，只需要借助标准C的内存管理函数realloc，我们就可以轻易改变数组的长度。函数realloc是比较费时的，如果每插入/删除一个元素就要调用realloc一次，不但会带来性能的下降，而且可能造成内存碎片。为了解决这个问题，需要使用一个称为预先分配的惯用手法。预先分配实际上也是用空间换时间的典型应用，下面我们看看它的实现。

### 扩展空间

在扩展数组时，不是一次扩展一个元素，而是一次扩展多个元素。至于应该扩展多少个，根据以往的经验大概是扩展为现有元素个数的1.5倍。

```
#define MIN_PRE_ALLOCATE_NR 10
static Ret darray_expand(DArray* thiz, size_t need)
{
        return_val_if_fail(thiz != NULL, RET_INVALID_PARAMS);

        if((thiz->size + need) > thiz->alloc_size)
        {
            size_t alloc_size = thiz->alloc_size + (thiz->alloc_size>>1)
                              + MIN_PRE_ALLOCATE_NR;

            void** data = (void**)realloc(thiz->data, sizeof(void*) *   alloc_size);
            if(data != NULL)
            {
                thiz->data = data;
                thiz->alloc_size = alloc_size;
            }
        }

        return ((thiz->size + need) <= thiz->alloc_size) ? RET_OK : RET_FAIL;
}

Ret darray_insert(DArray* thiz, size_t index, void* data)
```

```
{
        Ret ret = RET_OOM;
        size_t cursor = index;
        return_val_if_fail(thiz != NULL, RET_INVALID_PARAMS);

        cursor = cursor < thiz->size ? cursor : thiz->size;

        if(darray_expand(thiz, 1) == RET_OK)
        {
            size_t i = 0;
            for(i = thiz->size; i > cursor; i--)
            {
                thiz->data[i] = thiz->data[i-1];
            }

            thiz->data[cursor] = data;
            thiz->size++;

            ret = RET_OK;
        }

        return ret;
}
```

扩展的大小由以下公式得出。

```
size_t alloc_size = thiz->alloc_size + (thiz->alloc_size>>1) + MIN_PRE_ALLOCATE_NR;
```

在计算1.5*thiz->alloc_size时，我们实际上并不使用1.5 * thiz->alloc_size，因为这样会带来浮点数计算。大多数嵌入式平台并不支持硬件浮点数计算，浮点数的计算要比定点数的计算慢上百倍。

我们也不使用thiz->alloc_size + thiz->alloc_size/2，因为如果编译器不做优化，除法指令也是比较慢的操作。特别是像在ARM这种没有除法指令的芯片中，需要很多条指令才能实现除法的计算。

这里我们使用了thiz->alloc_size + (thiz->alloc_size>>1)，这是最快的方法。后面加上MIN_PRE_ALLOCATE_NR的原因是避免thiz->alloc_size为0时存在的错误。

**减小空间**

在删除元素时也不是马上释放空闲空间，而是等到空闲空间高于某个值时才释放它们。这里我们的做法是：当空闲空间多于有效空间时，将总空间调整为有效空间的1.5倍。

```
static Ret darray_shrink(DArray* thiz)
{
        return_val_if_fail(thiz != NULL, RET_INVALID_PARAMS);

        if((thiz->size < (thiz->alloc_size >> 1)) &&
```

```
        (thiz->alloc_size >      MIN_PRE_ALLOCATE_NR))
        {
            size_t alloc_size = thiz->size + (thiz->size >> 1);

            void** data = (void**)realloc(thiz->data, sizeof(void*) *   alloc_size);
            if(data != NULL)
            {
                thiz->data = data;
                thiz->alloc_size = alloc_size;
            }
        }

        return RET_OK;
}

Ret darray_delete(DArray* thiz, size_t index)
{
        size_t i = 0;
        Ret ret = RET_OK;

        return_val_if_fail(thiz != NULL && thiz->size > index,   RET_INVALID_PARAMS);

        darray_destroy_data(thiz, thiz->data[index]);
        for(i = index; (i+1) < thiz->size; i++)
        {
            thiz->data[i] = thiz->data[i+1];
        }
        thiz->size--;

        darray_shrink(thiz);

        return RET_OK;
}
```

为了避免极端情况下频繁resize，在总空间小于或等于MIN_PRE_ALLOCATE_NR时，我们并不减少空间的大小。

## 3.2 排序

大多数高级排序算法都是针对数组实现的，接下来我们一起来学习以下几种排序算法，学习算法本身只是我们的目标之一，最重要的是要从中学习一些思考问题的方法。对比不同算法的特点，也有助于我们在设计时做出正确的选择。

这里我们请读者实现冒泡排序、快速排序和归并排序。要求如下。

(1) 算法应该同时支持升序排序和降序排序。在升序排列中，前面的元素总是小于或等于后面的元素；在降序排序中，前面的元素总是大于或等于后面的元素。

(2) 算法应该同时支持多种数据类型。教科书上都是以整数排序为示例的，这种简化有利于

学生将精力集中在算法本身之上。但在现实中，我们不能只满足于了解算法本身，而是要写出一些具有实用价值的程序来。

对于前面提的两点要求——“算法应该同时支持升序排序和降序排序”和“算法应该同时支持多种数据类型”，如果是认真阅读过前面章节的读者，应该马上会想到利用回调函数。

这样想就对了！软件设计的关键在于熟能生巧，我们反复练习这些基本技巧也意在于此。熟到凭本能就可以运用正确的方法时，那也离所谓的高手不远了。言归正传，我们已经定义过比较回调函数的原型了。

```
typedef int (*DataCompareFunc)(void* ctx, void* data);
```

我们先实现一个整数的比较函数，升序比较函数的实现如下。

```
int int_cmp(void* a, void* b)
{
        return (int)a - (int)b;
}
```

降序比较函数的实现如下。

```
int int_cmp_invert(void* a, void* b)
{
        return (int)b - (int)a;
}
```

比较函数不依赖于具体的数据类型，这样我们就把算法的变化部分独立出来了。是升序还是降序完全由回调函数决定。下面我们看看算法的具体实现。

## ▶冒泡排序

冒泡排序的名字很形象地表达了算法的原理，对于降序排列时，可以认为排序过程是轻的物体（值较小的元素）不断往上浮的过程。不过对于升序排序，将其视作重的物体（值较大的元素）不断向下沉的过程更为合适。升序排列更符合人类的思考方式，因此这里我们是按升序排序来实现冒泡排序的（通过使用不同的比较函数，同样可以支持降序排序）。

```
Ret bubble_sort(void** array, size_t nr, DataCompareFunc cmp)
{
        size_t i     = 0;
        size_t max   = 0;
        size_t right = 0;

        return_val_if_fail(array != NULL && cmp != NULL, RET_INVALID_PARAMS);

        if(nr < 2)
        {
```

```
            return RET_OK;
        }

        for(right = nr - 1; right > 0; right--)
        {
            for(i = 1, max = 0; i < right; i++)
            {
                if(cmp(array[i], array[max]) > 0)
                {
                max = i;
                }
            }

            if(cmp(array[max], array[right]) > 0)
            {
                void* data = array[right];
                array[right] = array[max];
                array[max] = data;
            }
        }

        return RET_OK;
}
```

冒泡排序是最简单直观的排序算法，教科书上通常将其作为第一个排序算法来讲。从性能上来看，与其他高级排序算法相比，它似乎没有存在的理由。除了教学目的之外，它是否有实际价值呢？答案是有的，原因有以下两点。

(1) 实现简单，而简单的程序通常更可靠。虽然我很多年没有写过冒泡排序算法了，但这次从写代码、编译，再到测试，都是一次性通过的。而我在写快速排序时却出了好几次错误，而且最后参考了教科书才完成。如果非要自己动手写排序算法时，我首先肯定会想到写一个冒泡排序算法，直到一定需要更快的算法时才会考虑其他算法。

(2) 在数据量不大时，所有排序算法性能差别不大。有文章指出，高级排序算法只有在元素个数多于1000时，性能才出现显著提升。在90%的情况下，我们存储的元素个数只有几十到上百个而已，比如进程数、窗口个数和配置信息等的数量通常都不会很大，冒泡排序其实是更好的选择。

请记住：在完成同样任务的情况下，越简单越好。

同时还要强调侯捷先生的那句话：学从难处学，用从易处用。

### ▶快速排序

快速排序当然是以其性能优异出名了，而且它不需要额外的空间。如果数据量大而且数据全部存放在内存中，那么快速排序是首选的排序方法。具体的排序过程是先将元素分成两个区，所有小于某个元素的值在第一个区，其他元素在第二区。然后分别对这两个区进行快速排序，直到所分的区只剩下一个元素为止。

```
void quick_sort_impl(void** array, size_t left, size_t right, DataCompareFunc cmp)
{
        size_t save_left  = left;
        size_t save_right = right;
        void* x = array[left];
        /*这个循环，让小于x的元素在左边，大于x的元素在右边。*/
        while(left < right)
        {
            while(cmp(array[right], x) >= 0 && left < right) right--;
            if(left != right)
            {
                array[left] = array[right];
                left++;
            }

            while(cmp(array[left], x) <= 0 && left < right) left++;
            if(left != right)
            {
                array[right] = array[left];
                right--;
            }
        }
        array[left] = x;
        /*对左半部分排序*/
        if(save_left < left)
        {
            quick_sort_impl(array, save_left, left-1, cmp);
        }
        /*对右半部分排序*/
        if(save_right > left)
        {
            quick_sort_impl(array, left+1, save_right, cmp);
        }

        return;
}

Ret quick_sort(void** array, size_t nr, DataCompareFunc cmp)
{
        Ret ret = RET_OK;

        return_val_if_fail(array != NULL && cmp != NULL, RET_INVALID_PARAMS);

        if(nr > 1)
        {
            quick_sort_impl(array, 0, nr - 1, cmp);
        }

        return ret;
}
```

战胜软件复杂度是“系统程序员成长计划”的中心思想之一。战胜软件复杂度包括防止复杂度增长和降低复杂度两个方面。降低复杂度的方法主要有抽象和分而治之两种，快速排序是分而治之的具体体现。

## 归并排序

与快速排序一样，归并排序也是分而治之的应用。不同的是，它先让左右两部分进行排序，然后把它们合并起来。在排序左右两部分时，同样使用归并排序。快速排序可以看作是自顶向下的方法，而归并排序可以看作是自底向上的方法。

```
static Ret merge_sort_impl(void** storage, void** array, size_t low, size_t mid,
                           size_t high, DataCompareFunc cmp)
{
        size_t i = low;
        size_t j = low;
        size_t k = mid;
        /*对左半部分排序*/
        if((low + 1) < mid)
        {
            size_t x = low + ((mid - low) >> 1);
            merge_sort_impl(storage, array, low, x, mid, cmp);
        }
        /*对右半部分排序*/
        if((mid + 1) < high)
        {
            size_t x = mid + ((high - mid) >> 1);
            merge_sort_impl(storage, array, mid, x, high, cmp);
        }
        /*合并两个有序数组*/
        while(j < mid && k < high)
        {
            if(cmp(array[j], array[k]) <= 0)
            {
                storage[i++] = array[j++];
            }
            else
            {
                storage[i++] = array[k++];
            }
        }

        while(j < mid)
        {
            storage[i++] = array[j++];
        }

        while(k < high)
        {
            storage[i++] = array[k++];
        }

        for(i = low; i < high; i++)
        {
            array[i] = storage[i];
        }

        return RET_OK;
```

```
}

Ret merge_sort(void** array, size_t nr, DataCompareFunc cmp)
{
        void** storage = NULL;
        Ret ret = RET_OK;

        return_val_if_fail(array != NULL && cmp != NULL, RET_INVALID_PARAMS);

        if(nr > 1)
    {
        /*先分配一块辅助空间*/
            storage = (void**)malloc(sizeof(void*) * nr);
            if(storage != NULL)
            {
                ret = merge_sort_impl(storage, array, 0, nr>>1, nr, cmp);

                free(storage);
            }
        }

        return ret;
}
```

归并排序需要额外的存储空间，这部分空间和被排序的数组所占空间一样大。大部分示例代码里，都会在每次递归调用中分配空间，这样做会使性能下降。这里我们选择了事先分配一块空间，在排序过程中重复使用，算法更简单，性能也得到提高。

因为会把要排序的数组分成*N*个部分，所以可以把归并排序称为*N*路排序。上面实现的归并排序实际是归并算法的一个特例：*两路归并*。看似归并排序与快速排序相比几乎没有优势可言，但是归并排序更重要的能力在于处理大量数据的排序，它不要求被排序的数据全部在内存中，所以在数据大于内存的容纳能力时，归并排序就能大展身手了。归并排序最常用的地方是*数据库管理系统*（DBMS），因为数据库中存储的数据通常无法全部加载到内存中来的。有兴趣的读者可以阅读相关资料。

## ▶排序算法的测试

虽然排序算法的实现各有不同，但它们的目的都一样：让数据处于有序状态。所以在写自动测试时，没有必要为每一种算法都写一个测试程序。通过将排序算法作为回调函数传入，我们可以共用一个测试程序。

```
static void** create_int_array(int n)
{
        int i = 0;
        int* array = (int*)malloc(sizeof(int) * n);

        for(i = 0; i < n; i++)
        {
```

```
        array[i] = rand();
    }

    return (void**)array;
}

static void sort_test_one_asc(SortFunc sort, int n)
{
    int i = 0;
    void** array = create_int_array(n);

    sort(array, n, int_cmp);
    /*检查数据是否有序排列*/
    for(i = 1; i < n; i++)
    {
        assert(array[i] >= array[i-1]);
    }

    free(array);

    return;
}

void sort_test(SortFunc sort)
{
    int i = 0;
    for(i = 0; i < 1000; i++)
    {
        sort_test_one_asc(sort, i);
    }

    return ;
}
```

### 将排序算法集成到动态数组中

把排序算法放到动态数组里面并不合适，原因在于：

(1) 绑定动态数组与特定算法不如让用户根据需要自行选择；
(2) 在动态数组中实现排序算法不利于算法的重用。

所以我们给动态数组增加一个排序函数，但排序算法通过回调函数传入。

```
Ret  darray_sort(DArray* thiz, SortFunc sort, DataCompareFunc cmp);
```

## 3.3 有序数组的两个应用

前面我们学习了数组的排序方法，通常我们对数组排序不是为了排序而排序，而是有其他的目的，这里了解一下有序数组的两个常见应用。

## 二分查找

二分查找也称为折半查找，它的前提是数组中的元素是有序的。算法过程如下(假定数组为升序)：先拿要查找的元素与数组中间位置的元素相比较，如果要找的元素小则在数组的前半部分查找，大则在数组的后半部分查找，相等则说明找到了。重复这个过程直到找到或者数组被分成单个元素为止。实现如下所示。

```
int qsearch(void** array, size_t nr, void* data, DataCompareFunc cmp)
{
        int low    = 0;
        int mid    = 0;
        int high   = nr-1;
        int result = 0;

        return_val_if_fail(array != NULL && cmp != NULL, -1);

        while(low <= high)
        {
            mid  = low + ((high - low) >> 1);
            result = cmp(array[mid], data);

            if(result == 0)
            {
                return mid;
            }
            else if(result < 0)
            {
                low = mid + 1;
            }
            else
            {
                high = mid - 1;
            }
        }

        return -1;
}
```

在编写二分查找的代码时，除了算法本身外还要注意两个问题。

(1) 计算中间位置的方法。我在这里用了`mid = low + ((high - low) >> 1)`而不是`(low + high)/2`，目的是为了避免整数溢出和除法计算。

(2) 边界值问题。在编写排序和查找的程序时，最容易犯边界值错误。写程序时一定要保持思路清晰，不妨自己在大脑里模拟一下计算机执行你写的程序的过程，使用不同的输入，并观察所得的结果，最后再加上自动测试，就可以大大减少出错的概率。

## 去除重复元素

在工作中，我经常使用Linux中的命令sort和uniq的组合。uniq的功能是去除重复的元素，

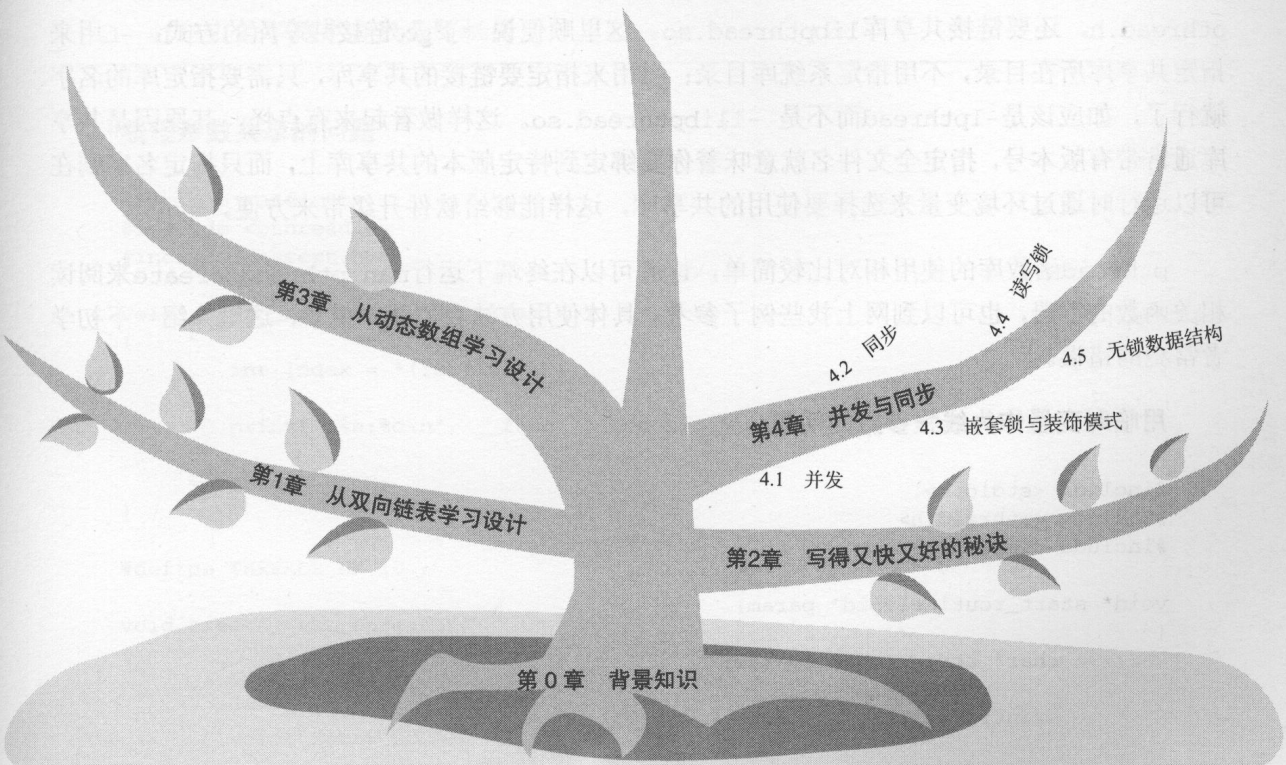
它的前提也是要求数据是有序的。下面我们写一个程序，它将打印数组中不重复元素（整数）。

```
Ret  unique_print_int(void* ctx, void* data)
{
        if(*(int*)ctx != (int)data)
        {
            *(int*)ctx = (int)data;
            printf("%d ", (int)data);
        }

        return RET_OK;
}

    darray_foreach(darray, unique_print_int, &data);
```

**注意** 记得把data初始化成不等于第一个元素的值，否则可能漏打第一个元素。这个算法当然同样适用链表，只要是有序的即可。

# 第 4 章

# 并发与同步


第3章 从动态数组学习设计
第1章 从双向链表学习设计
第0章 背景知识
第2章 写得又快又好的秘诀
第4章 并发与同步
4.1 并发
4.2 同步
4.3 嵌套锁与装饰模式
4.4 读写锁
4.5 无锁数据结构

## 4.1 并发

这几年并发技术受到前所未有的关注：CPU进入多核时代，连手机芯片都使用三核的CPU了（AP、BP和DSP都集成到一颗芯片上）。天生具有并发能力的语言Erlang逐渐成为热点。网格和云计算开始进入实用阶段。还有一些新技术更是让我闻所未闻，初学者也大可不必被这些铺天盖地的新名词吓倒。根据我的经验来看，这些技术或许能够改变产业的格局，对人类生活造成重大影响，但从实现角度来看并无多少革新，相反，大部分都是传统技术的继承和改进。这几年我一直在研究开源的基础软件，实际上我没有发现多少“新”东西或者核心技术。要说真正的核心技术还是如序言中说的：战胜复杂度和应对变化。

作为系统程序员，掌握基础理论和经典的设计方法，比去追逐一些所谓的新技术要实用得多，基础打扎实了，学习新知识也是很容易的事。在接下来几节中，我们一起来学习传统的并发编程知识。在这里我们请读者完成下列任务。

了解Linux下多线程编程的基本方法，以双向链表为载体实现传统的生产者-消费者模型：用一个线程往双向链表中添加元素，用另一个线程从这里面取出。

Linux下的多线程编程使用pthread（POSIX Thread）函数库，使用时需要包含头文件pthread.h，还要链接共享库libpthread.so。这里顺便说一下gcc链接共享库的方式：-L用来指定共享库所在目录，不用指定系统库目录；-l用来指定要链接的共享库，只需要指定库的名字就行了，如应该是-lpthread而不是 -llibpthread.so。这样做看起来有点怪，其原因是共享库通常带有版本号，指定全文件名就意味着你要绑定到特定版本的共享库上，而只指定名字则在可以运行时通过环境变量来选择要使用的共享库，这样能够给软件升级带来方便。

pthread函数库的使用相对比较简单，读者可以在终端下运行man pthread_create来阅读相关函数的手册，也可以到网上找些例子参考。具体使用方法我们就不讲了，这里介绍一下初学者常犯的错误。

**用临时变量作为线程参数的问题**

```
#include <stdio.h>
#include <pthread.h>
#include <assert.h>

void* start_routine(void* param)
{
        char* str = (char*)param;

        printf("%s:%s\n", __func__, str);
```

```
        return NULL;
}

pthread_t create_test_thread()
{
        pthread_t id = 0;
        char str[] = "it is ok!";

        pthread_create(&id, NULL, start_routine, str);

        return id;
}

int main(int argc, char* argv[])
{
        void* ret = NULL;

        pthread_t id = create_test_thread();

        pthread_join(id, &ret);

        return 0;
}
```

**分析** 由于新线程和当前线程是并发的，因此无法预测谁先谁后。可能create_test_thread已经执行完了，str也已经被释放了，新线程才得到这参数，此时它的内容已经无法确定了，打印出的字符串自然是随机的。

### 线程参数共享的问题

```
#include <stdio.h>
#include <pthread.h>
#include <assert.h>

void* start_routine(void* param)
{
        int index = *(int*)param;

        printf("%s:%d\n", __func__, index);

        return NULL;
}

#define THREADS_NR 10

void create_test_threads()
{
        int i = 0;

        void* ret = NULL;
```

```
        pthread_t ids[THREADS_NR] = {0};

        for(i = 0; i < THREADS_NR; i++)
        {
            pthread_create(ids + i, NULL, start_routine, &i);
        }

        for(i = 0; i < THREADS_NR; i++)
        {

            pthread_join(ids[i], &ret);
        }

        return ;
}

int main(int argc, char* argv[])
{
        create_test_threads();

        return 0;
}
```

**分析**　由于新线程和当前线程是并发的，因此无法预测谁先谁后。i在不断变化，所以新线程得到的参数值是无法预知的，打印出的字符串自然也是随机的。

### 虚假并发

```
#include <stdio.h>
#include <pthread.h>
#include <assert.h>

void* start_routine(void* param)
{

        int index = *(int*)param;

        printf("%s:%d\n", __func__, index);

        return NULL;
}

#define THREADS_NR 10

void create_test_threads()
{
        int i = 0;
        void* ret = NULL;

        pthread_t ids[THREADS_NR] = {0};
```

```
        for(i = 0; i < THREADS_NR; i++)
        {
            pthread_create(ids + i, NULL, start_routine, &i);
            pthread_join(ids[i], &ret);
        }

        return ;
}

int main(int argc, char* argv[])
{
        create_test_threads();

        return 0;
}
```

**分析** 因为pthread_join会阻塞其他线程直到当前线程退出，所以这些线程实际上是串行执行的，一个退出了，才创建下一个。当年一个同事写了一个这样的多线程测试程序，结果就是没有测试出一个潜伏的问题，直到产品运行时，这个问题才暴露出来。



访问线程共享的数据时要加锁，让访问串行化，否则就会出问题。比如，当你正在访问双向链表的某个结点时，它可能已经被另外一个线程删掉了。加锁的方式有很多种，像互斥锁（mutex=mutual exclusive lock）、信号量（semaphore）和自旋锁（spin lock）等都是常用的，它们的使用同样很简单，我们就不多说了。

在加锁/解锁时，初学者常犯两类错误。

一类错误是存在没有解锁的路径。初学者常见的做法就是，进入某个临界函数时加锁，在函数结尾的地方解锁，我甚至见过这种写法。

```
{
    /*这里加锁*/
    ...
        return...;
    /*这里解锁*/
}
```

如果你也犯了这种错误，那么请好好反思一下。有时候，return的地方太多，在某一处忘记解锁是可能的，就像内存泄露一样，只是忘记解锁的后果更严重。像下面这个例子。

```
Ret dlist_insert(DList* thiz, size_t index, void* data)
{
        DListNode* node = NULL;
        DListNode* cursor = NULL;

        return_val_if_fail(thiz != NULL, RET_INVALID_PARAMS);
```

```
    dlist_lock(thiz);

    if((node = dlist_create_node(thiz, data)) == NULL)
    {
        dlist_unlock(thiz);
        return RET_OOM;
    }

    if(thiz->first == NULL)
    {
        thiz->first = node;

        dlist_unlock(thiz);
        return RET_OK;
    }
    ...

    dlist_unlock(thiz);

    return RET_OK;
}
```

如果一个函数有五六个甚至更多的地方需要返回，那么遗忘一两处也是很常见的，即使没有忘记，在每个返回的地方都要去解锁和释放相关资源也是很麻烦的。在这种情况下，我们最好是实现单入口单出口的函数，常见的做法有以下两种。

(1) 使用goto语句。在Linux内核里大量使用了该方式。示例如下。

```
Ret dlist_insert(DList* thiz, size_t index, void* data)
{
    Ret ret = RET_OK;
    DListNode* node = NULL;
    DListNode* cursor = NULL;

    return_val_if_fail(thiz != NULL, RET_INVALID_PARAMS);

    dlist_lock(thiz);

    if((node = dlist_create_node(thiz, data)) == NULL)
    {
        ret = RET_OOM;
        goto done;
    }

    if(thiz->first == NULL)
    {
        thiz->first = node;

        goto done;
    }
    ...
```

```
    done:
        dlist_unlock(thiz);

        return ret;
}
```

(2) 使用do{}while(0);语句。由于受到教科书的影响(不要用会破坏程序结构的goto语句),我习惯了这种做法。示例如下。

```
Ret dlist_insert(DList* thiz, size_t index, void* data)
{
        Ret ret = RET_OK;
        DListNode* node = NULL;
        DListNode* cursor = NULL;

        return_val_if_fail(thiz != NULL, RET_INVALID_PARAMS);

        dlist_lock(thiz);

        do
        {
            if((node = dlist_create_node(thiz, data)) == NULL)
            {
                ret = RET_OOM;
                break;
            }

            if(thiz->first == NULL)
            {
                thiz->first = node;
                break;
            }
        ...
        }while(0);

        dlist_unlock(thiz);

        return ret;
}
```

另一类错误是加锁顺序的问题。有时候为了提高效率，常常降低加锁的粒度，访问时不是用一个锁锁住整个数据结构，而是用多个锁来控制数据结构的各个部分。这样一来，当一个线程访问数据结构的某个部分时，另外一个线程还可以访问数据结构的其他部分。但是在有的情况下，你需要同时锁几个锁，这时就要注意了：所有线程一定要按相同的顺序加锁，按相反的顺序解锁。否则就可能出现死锁，即两个线程都拿着对方需要的锁，却又互相等待对方先释放锁的情况。

## 4.2 同步

在“生产者-消费者”的练习中，大部分人选择了由调用者来加锁：作为生产者，往双向链

表里插入数据时，先加锁，接着插入数据，然后解锁；作为消费者，从双向链表里取数据时，先加锁，接着删除数据，然后解锁。这是合理的，不过有点麻烦：每个调用者都要做这些动作，如果其中一个调用者忘记了解锁的步骤，就会造成死锁。而且调用者必须明确自己是在多线程下工作，这些代码放到单线程的环境中就不能使用了。

在很多情况下，由实现者来加锁是比较好的选择，那样对调用者更为友好，可以避免出现一些不必要的错误。比如像目前Linux下流行的DBUS，它是一套进程间通信框架，它同时拥有支持单线程和多线程的版本，但调用者不需要明确如何加锁/解锁，也不需要连接不同的库或用宏来进行控制，单线程版本和多线程版本的不同只是在一个初始化函数上。

这里我们请读者对前面实现的双向链表做点改进。

(1) 支持多线程和单线程的版本。对于多线程版本，由实现者（在链表）加锁/解锁，对于单线程版本，其性能不受影响（或很小）。

(2) 区分单线程版本和多线程版本时，既不需要链接不同的库，也不需要用宏来进行控制，完全可以在运行时切换。

(3) 保持双向链表的通用性，不依赖于特定的平台。

面对这个需求，一些初学者可能有点蒙了。以前在学校的时候，对于课本后面的练习，我总是信心百倍，原因很简单，我确信这些练习不管它的出现方式有多么不同，但总是与前面学过的知识有关。记得《如何求解问题：现代启发式方法》[①]中说过，正是这种练习的方式妨碍了我们提升解决问题的能力，在现实中解决问题时我们通常没有这么幸运。在本书中，我之所以把练习放在前面，就是为了刺激读者去思考，希望读者在学习知识的同时学习解决问题的方法。

这里我们应该怎么分析呢？要在双向链表里加锁，第一是要区分单线程和多线程，要链接同一个库，而且不能用宏来控制。第二是不能依赖于特定平台，但锁本身恰恰是依赖于平台的。怎么办？很明显这两个需求都要求锁的实现可以变化：单线程版本中，它什么都不做；多线程版本中，不同的平台有不同的实现。

我们要做的就是隔离变化。应该怎样隔离变化？前面我们已经练习过几次用回调函数来隔离变化，所有的读者应该都会想到这个方法，因为锁无非是具有两个功能——加锁和解锁，我们只要把它抽象成两个回调函数就行了。

这种方法是可行的。不过这里的情况与前面相比有点特殊。前面的回调函数都是些独立功能的函数，每个回调函数都有自己独立的上下文，而这里的多个回调函数具有相关的功能，并且共享同一个上下文（锁）；其次是这里的上下文（锁）是一个对象，有自己的生命周期，在完成自己的使命后就应该被销毁。

① 已由中国水利水电出版社出版。——编者注

这里我们引入接口（interface）这个术语，接口其实就是一个抽象的概念，它只定义调用者和实现者之间的契约，而不规定具体的实现方法。比如这里的锁就是一个抽象的概念，它有加锁/解锁两个功能，这就是调用者和实现者之间的契约。但光有这个概念不能做任何事情，只有具体的锁才能被使用。至于具体的锁，不同的平台有不同的实现，但调用者不用关心。正因为调用者不用关心接口的实现方法，接口成了隔离变化最有力的武器。

在这里，锁是一个接口，它有多种不同的实现方式，双向链表是锁的调用者，它不用关心锁的具体实现方法。通过接口，双向链表把锁的变化隔离开来：区分单线程和多线程，隔离平台相关性。在C语言中，接口的朴素定义是：一组相关的回调函数及其共享的上下文。我们看看锁这个接口怎么定义。

```
struct _Locker;
typedef struct _Locker Locker;

typedef Ret  (*LockerLockFunc)(Locker* thiz);
typedef Ret  (*LockerUnlockFunc)(Locker* thiz);
typedef void (*LockerDestroyFunc)(Locker* thiz);

struct _Locker
{
        LockerLockFunc    lock;
        LockerUnlockFunc  unlock;
        LockerDestroyFunc destroy;

        char priv[0];
};
```

这里要注意以下三个问题。

第一，接口一定要足够抽象，不能依赖任何具体实现的数据类型。接口一旦与某个具体实现关联了，另外一种实现就会遇到麻烦。比如这里你使用了pthread_mutex_t，那你要实现一个Win32下的锁怎么办呢。

第二，接口不能有create函数，但一定要有destroy函数。我们说过对象有自己的生命周期，创建它，使用它，然后销毁它。但接口只是一个概念，不可能通过这个概念凭空创建一个对象出来，对象只能通过具体实现来创建，所以接口不应该出现create自己的函数。一旦对象被创建出来，使用者应该在不再需要它时销毁它，在销毁对象时，如果还要知道它的实现方式才能销毁它，那就造成了调用者和实现者之间不必要的耦合，因此接口都要提供一个destroy函数，调用者可以直接销毁它。

第三，这里的priv用来存放上下文信息，也就是具体实现需要用到的数据结构。像前面的回调函数一样，我们可以用一个void* ctx的成员来保存上下文信息。我们使用的char priv[0];技巧，有点额外的好处：只需要一次内存分配，而且可以分配刚好够用的长度（0到任意长度）。

前面我们使用回调函数，调用时要判断回调函数是否为空，每个地方都要重复这个动作，让调用者使用不方便，所以我们把这些判断集中起来好了。

```
static inline Ret locker_lock(Locker* thiz)
{
        return_val_if_fail(thiz != NULL && thiz->lock != NULL,   RET_INVALID_PARAMS);

        return thiz->lock(thiz);
}

static inline Ret locker_unlock(Locker* thiz)
{
        return_val_if_fail(thiz != NULL && thiz->unlock != NULL, RET_INVALID_PARAMS);

        return thiz->unlock(thiz);
}

static inline void locker_destroy(Locker* thiz)
{
        return_if_fail(thiz != NULL && thiz->destroy != NULL);

        thiz->destroy(thiz);

        return;
}
```

下面我们来看看基于pthread_mutex的实现。

(1) 在locker_pthread.h中，提供一个创建函数。

```
Locker* locker_pthread_create(void);
```

(2) 在locker_pthread.c中，实现一些回调函数。

首先定义私有数据结构。

```
typedef struct _PrivInfo
{
        pthread_mutex_t mutex;
}PrivInfo;
```

然后创建对象。

```
Locker* locker_pthread_create(void)
{
        Locker* thiz = (Locker*)malloc(sizeof(Locker) + sizeof(PrivInfo));

        if(thiz != NULL)
        {
            PrivInfo* priv = (PrivInfo*)thiz->priv;
            /*让函数指针指到具体的函数上*/
```

```
        thiz->lock    = locker_pthread_lock;
        thiz->unlock  = locker_pthread_unlock;
        thiz->destroy = locker_pthread_destroy;

        pthread_mutex_init(&(priv->mutex), NULL);
    }

    return thiz;
}
```

最后实现以下几个回调函数。

```
/*加锁函数*/
static Ret  locker_pthread_lock(Locker* thiz)
{
    PrivInfo* priv = (PrivInfo*)thiz->priv;
    int ret = pthread_mutex_lock(&priv->mutex);
    return ret == 0 ? RET_OK : RET_FAIL;
}
/*解锁函数*/
static Ret locker_pthread_unlock(Locker* thiz)
{
    PrivInfo* priv = (PrivInfo*)thiz->priv;
    int ret = pthread_mutex_unlock(&priv->mutex);
    return ret == 0 ? RET_OK : RET_FAIL;
}
/*销毁函数*/
static void locker_pthread_destroy(Locker* thiz)
{
    PrivInfo* priv = (PrivInfo*)thiz->priv;
    int ret = pthread_mutex_destroy(&priv->mutex);
    SAFE_FREE(thiz);
  return;
}
```

我简单说一下这里面的几个问题。

(1) `malloc(sizeof(Locker) + sizeof(PrivInfo));` 前面的`char priv[0]`并不占空间，这是C语言新标准定义的，用于实现变长的缓冲区，在这里，它的长度是由`sizeof(PrivInfo)`决定的。

(2) `PrivInfo* priv = (PrivInfo*)thiz->priv;` 这里的`thiz->priv`只是一个定位符，实际上等于`(size_t)thiz + sizeof(Locker)`，帮我们定位到私有数据的内存地址上。

锁的使用方法如下。

(1) 单线程版本：

```
DList* dlist = dlist_create(NULL, NULL, locker_pthread_create());
```

(2) 多线程版本：

```
DList* dlist = dlist_create(NULL, NULL, NULL);
```

接口在软件设计中占有非常重要的地位，它是隔离变化和降低复杂度最有力的武器，几乎所有的设计模式都与接口有关。后面我们会反复加以练习，这里请读者先自行体会一下。

## 4.3 嵌套锁与装饰模式

在“生产者-消费者”模式的练习中，当由双向链表的实现者负责加锁时，一般都会遇到莫名其妙的死锁问题。有的读者可能已经查出来了，引起问题的原因是嵌套地加锁。比如在dlist_insert中调用了dlist_length，进入dlist_insert时已经加了一次锁，再调用dlist_length时又加了一次锁，这时就出现了死锁问题。

初学者遇到这个问题的时候，通常的做法是在调用dlist_length之前先解锁，调用完dlist_length后再重新加锁。这样做是存在问题的：一个原子操作变成几个原子操作，数据完整性就得不到保证了，在你重新加锁之前，其他线程可能利用这个空隙做了些别的事情。

有效解决这个问题的办法有两个，其一是实现一个内部版本的dlist_length，它在里面不加锁。其二是使用嵌套锁，即允许同一个线程多次加锁。pthread有嵌套锁的实现，不过我们在这里不用它，因为我们要提供一个更通用的解决方案。现在我们不再满足于实现一个双向链表，而是要实现一个跨平台的基础函数库。

在这里我们请读者实现一个嵌套锁，要求如下。

(1) 嵌套锁仍然兼容Locker接口。
(2) 嵌套锁的实现不依赖于特定平台。

### ▶嵌套锁的算法与实现

❑ 加锁算法。
(1) 如果没有任何线程加锁，就直接加锁，并且记录下当前线程的ID。
(2) 如果是当前线程加过锁了，就不用加锁了，只是将加锁的计数增加一。
(3) 如果其他线程加锁了，那就等待直到加锁成功，后继步骤与第一种情况相同。

❑ 解锁算法。
(1) 如果不是当前线程加的锁或者没有人加锁，那这是错误的调用，直接返回。
(2) 如果是当前线程加锁了，将加锁的计数减1。如果计数仍大于0，说明当前线程加了多次锁，直接返回就行了。如果计数为0，说明当前线程只加了一次锁，则执行解锁动作。

这个逻辑很简单，要做到兼容Locker的接口和平台无关，我们还需要引入装饰模式这个概

念。装饰模式的作用是在不改变对象本质（接口）的前提下，给对象添加附加的功能。和继承不同的是，它不是针对整个类的，而只是针对单个对象的。“装饰”这个名字非常直观地表现它的意义。就好比你在自己电脑的显示器上做了点装饰，比如贴上一张卡通画：第一，这么做并没有改变显示器的本质，显示器还是显示器；第二，只有你自己的显示器上多了张卡通画，其他显示器没有受到影响。

这里我们要对一把锁进行装饰，不改变它的接口，但给它加上嵌套的功能。下面我们看看在C语言里的实现方法。

**创建函数的原型**

由于获取当前线程ID的函数是平台相关的，我们要用回调函数来抽象它。

```
typedef int (*TaskSelfFunc)(void);
Locker* locker_nest_create(Locker* real_locker, TaskSelfFunc task_self);
```



这里可以看出：传入的是一把锁，返回的还是一把锁，没有改变接口，但是返回的锁已经具有嵌套调用的功能了。

**嵌套锁的具体实现**

私有信息：拥有锁的线程ID、加锁的计数，被装饰的锁和获取当前线程ID的回调函数。

```
typedef struct _PrivInfo
{
            int owner;
            int refcount;
            Locker* real_locker;
            TaskSelfFunc task_self;
}PrivInfo;
```

实现加锁函数：如果当前线程已经加锁，就只是增加加锁计数，否则就加锁。

```
static Ret  locker_nest_lock(Locker* thiz)
{
        Ret ret = RET_OK;
        PrivInfo* priv = (PrivInfo*)thiz->priv;
        /*检查当前线程是否已经加锁*/
        if(priv->owner == priv->task_self())
        {
            priv->refcount++;
        }
        else
    {
            /*加锁，然后记录当前线程ID*/
            if( (ret = locker_lock(priv->real_locker)) == RET_OK)
            {
```

```
                priv->refcount = 1;
                priv->owner = priv->task_self();
            }
        }

        return ret;
}
```

实现解锁函数：只有当前线程加的锁才能解锁，先减少加锁计数，计数为0时才真正解锁，否则直接返回。

```
static Ret  locker_nest_unlock(Locker* thiz)
{
        Ret ret = RET_OK;
        PrivInfo* priv = (PrivInfo*)thiz->priv;
        /*当前线程之前没有加锁，尝试解锁是错误的*/
        return_val_if_fail(priv->owner == priv->task_self(), RET_FAIL);

        /*减少引用计数，计数为0时，才真正解锁*/
    priv->refcount--;
        if(priv->refcount == 0)
        {
            priv->owner = 0;
            ret = locker_unlock(priv->real_locker);
        }

        return ret;
}
```

**嵌套锁的使用方法**

与普通的锁相比，除了创建方法稍有不同外，调用方法完全一样。

```
Locker* locker = locker_pthread_create();
Locker* nest_locker = locker_nest_create(locker,
(TaskSelfFunc)pthread_self);
DList*  dlist = dlist_create(NULL, NULL, nest_locker);
```

装饰模式最有用的地方在于，它可以在不影响调用者的前提下，给单个对象增加功能，即使是加了多级装饰，调用者也不用担心会受影响。

## 4.4 读写锁

在前面的实现中，像dlist_length这类的查询函数也要加锁，那样才能保证在查询过程中对象的状态不会被其他线程所改变。加锁虽然阻止了其他线程修改对象，但同时也阻止了其他线程查询对象。如果大多数情况下，线程只是查询对象的状态而不修改它，这种设计就不是一种高效的方法了，因为它不允许多个线程同时查询。我们能不能实现一种锁，它既能串行化对数据结

构的修改，同时又支持并行的查询呢？

这就是所谓的读写锁，也称为共享-互斥锁。这种锁在数据库管理系统（DBMS）和操作系统内核中大量应用，作为系统程序员，我们有必要了解它的实现机制。这里请读者先自行尝试实现读写锁，要求如下。

(1) 不依赖于特定平台。

(2) 在任何情况下都不带来额外的性能开销。

记住多想多练不要偷懒，学习知识点不是我们最重要的目标，知识点能帮你解决别人解决过的问题，但对你解决新问题未必有多大好处，真正的程序员不应当只是问题解决方案的贩卖者。不断从思考中参悟解决问题的方法，再加上灵活应用已经掌握的知识点，你的设计水平才会大大提高，这也是本书希望读者所能达到的目标。读写锁在加锁时，要区分是为了读而加锁，还是为了写而加锁，因此和递归锁不同的是，它无法兼容Locker接口。不过为了做到不依赖于特定平台，我们可以利用Locker的接口来抽象锁的实现。利用现有的锁来实现读写锁。读写锁的可变的部分已经被Locker隔离了，所以读写锁本身不需要做成接口，它只是一个普通对象而已。



```
struct _RwLocker;
typedef struct _RwLocker RwLocker;

/*创建一个读写锁*/
RwLocker* rw_locker_create(Locker* rw_locker, Locker* rd_locker);
/*加写锁*/
Ret rw_locker_wrlock(RwLocker* thiz);
/*加读锁*/
Ret rw_locker_rdlock(RwLocker* thiz);
/*解锁*/
Ret rw_locker_unlock(RwLocker* thiz);
/*销毁对象*/
void rw_locker_destroy(RwLocker* thiz);
```

下面来看看具体的实现。

### 创建读写锁

```
RwLocker* rw_locker_create(Locker* rw_locker, Locker* rd_locker)
{
        RwLocker* thiz = NULL;
        return_val_if_fail(rw_locker != NULL && rd_locker != NULL, NULL);

        thiz = (RwLocker*)malloc(sizeof(RwLocker));
        if(thiz != NULL)
        {
            thiz->readers = 0;
            thiz->mode = RW_LOCKER_NONE;
            thiz->rw_locker = rw_locker;
```

```
        thiz->rd_locker = rd_locker;
    }

    return thiz;
}
```

读写锁的基本要求是：写的时候不允许任何其他线程读或写，读的时候允许其他线程读，但不允许其他线程写。所以在实现时，写的时候一定要加锁，第一个读的线程要加锁，后面其他线程读时，只是增加锁的引用计数。在这里我们需要两个锁：一个锁用来保存被保护的对象，一个锁用来保护引用计数。

**加写锁**

```
Ret rw_locker_wrlock(RwLocker* thiz)
{
    Ret ret = RET_OK;
    return_val_if_fail(thiz != NULL, RET_INVALID_PARAMS);

    if((ret = locker_lock(thiz->rw_locker)) == RET_OK)
    {
        thiz->mode = RW_LOCKER_WR;
    }

    return ret;
}
```

加写锁很简单，直接加用来保护受保护对象的锁，然后修改锁的状态为已加写锁。后面其他的线程如果想写，就需要在这个锁上等待，即使是想读也要等待（见后面的内容）。

**加读锁**

```
Ret rw_locker_rdlock(RwLocker* thiz)
{
    Ret ret = RET_OK;
    return_val_if_fail(thiz != NULL, RET_INVALID_PARAMS);

    if((ret = locker_lock(thiz->rd_locker)) == RET_OK)
    {
        thiz->readers++;
        /*第一个加读锁的线程还要加写锁，即有人读则禁止写。*/
        if(thiz->readers == 1)
        {
            ret = locker_lock(thiz->rw_locker);
            thiz->mode = RW_LOCKER_RD;
        }
        locker_unlock(thiz->rd_locker);
    }

    return ret;
}
```

先尝试加用来保护引用计数的锁，将引用计数加1。如果当前线程是执行第一个读操作，就要去加用来保护受保护对象的锁。如果此时已经有线程在执行写操作，就一直等待，直到加锁成功，然后把锁的状态设置为已加读锁，最后再解开保护引用计数的锁。

**解锁**

```
Ret rw_locker_unlock(RwLocker* thiz)
{
        Ret ret = RET_OK;
        return_val_if_fail(thiz != NULL, RET_INVALID_PARAMS);

        if(thiz->mode == RW_LOCKER_WR)
    {
        /*解写锁*/
            thiz->mode == RW_LOCKER_NONE;
            ret = locker_unlock(thiz->rw_locker);
        }
        else
    {
        /*解读锁，要检查是否有其他线程在读。*/
            assert(thiz->mode == RW_LOCKER_RD);
            if((ret = locker_lock(thiz->rd_locker)) == RET_OK)
            {
                thiz->readers--;
                if(thiz->readers == 0)
                {
                    thiz->mode == RW_LOCKER_NONE;
                    ret = locker_unlock(thiz->rw_locker);
                }
                locker_unlock(thiz->rd_locker);
            }
        }

        return ret;
}
```

解锁时根据状态来决定，解写锁直接解用来保护受保护对象的锁。解读锁时，先要加用来保护引用计数的锁，将引用计数减1。直到最后一个读操作时，我们才解开用来保护受保护对象的锁，最后再解开用来保护引用计数的锁。

从上面所讲的读写锁的实现，我们可以看出，想让读写锁充分发挥作用，需要基于以下两个假设。

(1) 读写的不对称性，读的次数远远大于写的次数。像数据库就是这样，绝大部分时间是在查询，而修改的情况相对少得多，所以数据库通常使用读写锁。

(2) 处于临界区的时间比较长。从上面的实现来看，使用读写锁与使用普通锁相比，加/解锁的次数反而要多。如果处于临界区的时间比较短，比如和修改引用计数所需的时间差不多，那么

即使全部是读操作，读写锁的效率也会低于普通锁。

## 4.5 无锁数据结构

提到并行计算，我们通常都会想到加锁，事实却并非如此，大多数并发是不需要加锁的。比如在不同电脑上运行的代码编辑器，两者并发运行不需要加锁。在一台电脑上同时运行的媒体播放器和代码编辑器，两者并发运行不需要加锁（当然系统调用和进程调度是要加锁的）。在同一个进程中运行多个线程，如果各自处理独立的事情也不需要加锁（当然系统调用、进程调度和内存分配是要加锁的）。在以上这些情况里，各个并发实体之间没有共享数据，所以虽然并发运行但不需要加锁。

多线程并发运行时，虽然有共享数据，但如果所有线程只是读取共享数据而不修改它，也是不用加锁的，比如代码段就是每个线程都会读取的共享“数据”，但是它不用加锁。

排除了以上这些情况后，其实程序员所要关注的是这样的情况：多线程之间有共享数据，有的线程要修改这些共享数据，而有的线程要读取这些共享数据。这种情况正是本节中我们所要讨论的内容。

在并发的环境里，加锁可以保护共享的数据，但是加锁也会存在以下问题。

(1) 由于临界区无法并发运行，进入临界区就需要等待，加锁会带来效率的降低。
(2) 在复杂的情况下，很容易造成死锁，带来并发实体之间无止境的互相等待。
(3) 在中断/信号处理函数中是不能加锁的，这给并发处理带来困难。
(4) 优先级倒置造成实时系统不能正常工作。低优先级进程拿到高优先级进程需要的锁，高优先级进程得到CPU却得不到锁，而低优先级进程得到锁却得不到CPU，结果是高/低优先级的进程都无法运行，中等优先级的进程可能在疯狂运行。

由于并发与加锁（互斥）的矛盾关系，无锁数据结构自然成为程序员关注的焦点，这也是本节要介绍的。

### ▶CPU 提供的原子操作

大约在七八年前，我和同事们用Apache的Xerces来解析XML文件，奇怪的是多线程反而比单线程慢。同事们找了很久也没有找出原因，只是证实使用多进程代替多线程会快一个数量级，在Windows上他们就使用了多进程的方式。后来移植到Linux时候，我发现Xerces每创建一个结点都会去更新一些全局的统计信息（比如把结点的总数加1），而它是用pthread_mutex来实现互斥的。这就是问题所在：一个XML文档有数以万计的结点，以50个线程并发为例，每个线程解析一个XML文档，总共要进行上百万次的加锁/解锁，几乎所有线程都在等待，你说能快得了吗？

当时我知道Windows下有InterlockedIncrement之类的函数，它们利用CPU的一些特殊指令来保证对整数的基本操作是原子的。在查找了一些资料后，我发现Linux下也有类似的函数，后来我把所加的锁全都去掉，换成这些原子操作，结果速度比多进程运行还要快上几倍。下面我们看++和--的原子操作在IA架构上的实现。

```
#define ATOMIC_SMP_LOCK "lock ; "
typedef struct { volatile int counter; } atomic_t;

static __inline__ void atomic_inc(atomic_t *v)
{
        __asm__ __volatile__(
            ATOMIC_SMP_LOCK "incl %0"
            :"=m" (v->counter)
            :"m" (v->counter));
}

static __inline__ void atomic_dec(atomic_t *v)
{
        __asm__ __volatile__(
            ATOMIC_SMP_LOCK "decl %0"
            :"=m" (v->counter)
            :"m" (v->counter));
}
```

## 单入单出的循环队列

单入单出的循环队列是一种特殊情况，虽然特殊但是很实用，重要的是它不需要加锁。这里的单入是指只有一个线程向队列里追加数据（push），而单出则是指只有一个线程从队列里取数据（pop）。循环队列与普通队列相比，不同之处在于它的最大数据储存量是事先固定好的，不能动态增长。尽管有这些限制，它的应用还是相当广泛的。这里我们介绍一下它的实现。

数据结构定义如下。

```
typedef struct _FifoRing
{
        int r_cursor;/*读指针*/
        int w_cursor;/*写指针*/
        size_t length;/*FIFO的长度*/
        void* data[0];/*FIFO的存储区，长度由length决定*/
}FifoRing;
```

r_cursor指向队列头，用于取数据。w_cursor指向队列尾，用于追加数据。length表示队列的最大数据储存量，data表示存放的数据，[0]在这里表示变长的缓冲区（前面我们已经讲过）。

### 创建函数

```
FifoRing* fifo_ring_create(size_t length)
```

```
{
    FifoRing* thiz = NULL;

    return_val_if_fail(length > 1, NULL);

    thiz = (FifoRing*)malloc(sizeof(FifoRing) + length * sizeof(void*));

    if(thiz != NULL)
    {
        thiz->r_cursor = 0;
        thiz->w_cursor = 0;
        thiz->length   = length;
    }

    return thiz;
}
```

这里我们要求队列的长度大于1而不是大于0，为什么呢？除了长度为1的队列没有什么意义外，更重要的原因是如果队列头与队列尾重叠（r_cursor = = w_cursor），那到底表示的是满队列还是空队列？这个必须得搞清楚才行，上次一个同事犯了这个错误，让我们查了很久。这里我们认为队列头与队列尾重叠时表示队列为空，这与队列初始状态一致，后面在写的时候始终保留一个空位，避免队列头与队列尾重叠，这样便可以消除歧义了。

**追加数据**

```
Ret fifo_ring_push(FifoRing* thiz, void* data)
{
    int w_cursor = 0;
    Ret ret = RET_FAIL;
    return_val_if_fail(thiz != NULL, RET_FAIL);

    w_cursor = (thiz->w_cursor + 1) % thiz->length;

    if(w_cursor != thiz->r_cursor)
    {
        thiz->data[thiz->w_cursor] = data;
        thiz->w_cursor = w_cursor;

        ret = RET_OK;
    }

    return ret;
}
```

队列头和队列尾之间还有1个以上的空位时就追加数据，否则返回失败。

**取数据**

```
Ret fifo_ring_pop(FifoRing* thiz, void** data)
{
```

```
        Ret ret = RET_FAIL;
        return_val_if_fail(thiz != NULL && data != NULL, RET_FAIL);
/*读写指针不重叠则表示有数据*/
        if(thiz->r_cursor != thiz->w_cursor)
        {
            *data = thiz->data[thiz->r_cursor];
            thiz->r_cursor = (thiz->r_cursor + 1)%thiz->length;

            ret = RET_OK;
        }

        return ret;
}
```

队列头和队列尾不重叠表示队列不为空，取数据并移动队列头。

## 单写多读的无锁数据结构



单写表示只有一个线程去修改共享数据结构，多读表示有多个线程去读取共享数据结构。前面介绍的读写锁可以有效地解决这个问题，但更高效的办法是使用无锁数据结构。思路如下。

就像为了避免显示闪烁而使用双缓冲一样，我们使用两份数据结构，一份数据结构用于读取，所有线程都可以在不加锁的情况下读取这个数据结构。另外一份数据结构用于修改，由于只有一个线程会修改它，所以也不用加锁。

在修改之后，我们再交换读/写的两个函数结构，把另外一份也修改过来，这样两个数据结构就一致了。在交换时要保证没有线程在读取，所以我们还需要一个读线程的引用计数。现在我们看看怎么把前面写的双向链表改为单写多读的无锁数据结构。

为了保证交换是原子的，我们需要一个新的原子操作CAS（compare and swap，比较并交换）。

```
#define CAS(_a, _o, _n)                                                 \
({ __typeof__(_o) __o = _o;                                             \
        __asm__ __volatile__(                                           \
            "lock cmpxchg %3,%1"                                        \
            : "=a" (__o), "=m" (*(volatile unsigned int *)(_a))         \
            :  "0" (__o), "r" (_n) );                                   \
        __o;                                                            \
    })
```

### 数据结构的定义

```
typedef struct _SwmrDList
{
        atomic_t rd_index_and_ref;
        DList* dlists[2];
}SwmrDList;
```

两个链表，一个用于读一个用于写。rd_index_and_ref的最高字节记录用于读取的双向链表的索引，低24位用于记录读取线程的引用记数，最多支持16 777 216个线程同时读取，这应该是足够了，所以后面我们不考虑它溢出的情况。

### 读取操作

```
int swmr_dlist_find(SwmrDList* thiz, DListDataCompareFunc cmp, void* ctx)
{
        int ret = 0;
        return_val_if_fail(thiz != NULL && thiz->dlists != NULL, -1);
        /*操作用于读取的链表，只增加引用计数，不需要加锁*/
        atomic_inc(&(thiz->rd_index_and_ref));
        size_t rd_index = (thiz->rd_index_and_ref.counter>>24) & 0x1;
        ret = dlist_find(thiz->dlists[rd_index], cmp, ctx);
        atomic_dec(&(thiz->rd_index_and_ref));

        return ret;
}
```

### 修改操作

```
Ret swmr_dlist_insert(SwmrDList* thiz, size_t index, void* data)
{
        Ret ret = RET_FAIL;
        DList* wr_dlist = NULL;
    return_val_if_fail(thiz != NULL && thiz->dlists != NULL, ret);
    /*先操作用于修改的那个链表，然后交换*/
        size_t wr_index = !((thiz->rd_index_and_ref.counter>>24) & 0x1);
        if((ret = dlist_insert(thiz->dlists[wr_index], index, data)) == RET_OK)
        {
            int rd_index_old = thiz->rd_index_and_ref.counter & 0xFF000000;
            int rd_index_new = wr_index << 24;

            do
            {
                usleep(100);
            }while(CAS(&(thiz->rd_index_and_ref), rd_index_old, rd_index_new));

            wr_index = rd_index_old>>24;
            /*把另一个链表也修改过来*/
            ret = dlist_insert(thiz->dlists[wr_index], index, data);
        }

        return ret;
}
```

先修改用于修改操作的双向链表，修改完成之后等到没有线程读取时，交换读/写两个链表，再修改另一个链表，此时两个链表状态保持一致。

稍做改进，对修改的操作进行加锁，就可以支持多读多写的数据结构，读是无锁的，而写是

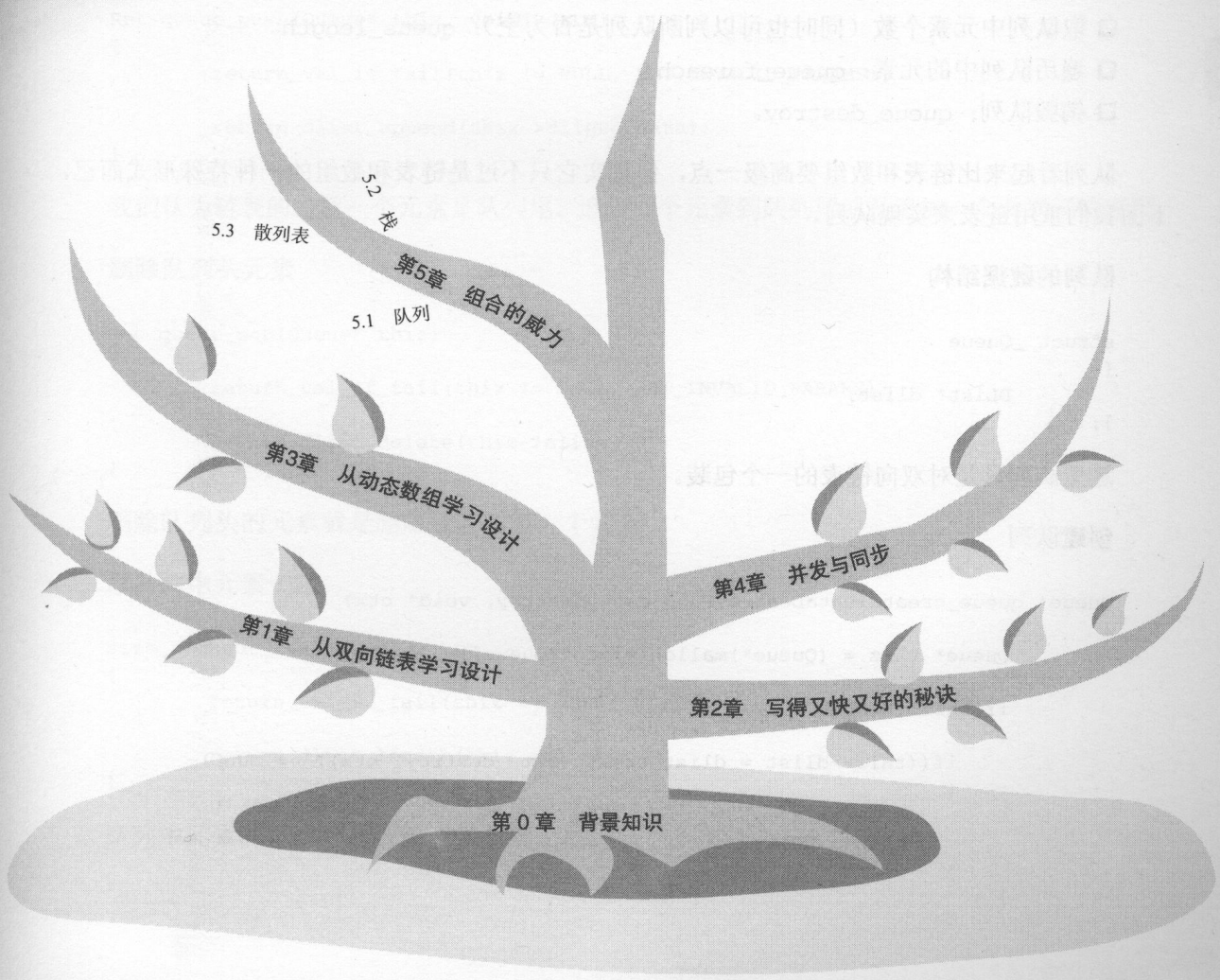
加锁的。

### 真正的无锁数据结构

Andrei Alexandrescu的“Lock-Free Data Structures”估计是这方面最经典的论文了。对他的方法我开始是感到惊奇，后来却感到失望了，惊奇的是算法的巧妙，失望的是无锁的限制和代价。作者最后说这种数据结构只适用于WRRMBNTM（Write-Rarely-Read-Many-But-Not-Too-Many，写操作很少、读操作很多但不至于太多）的情况。而且每次修改都要复制（甚至多次复制）整个数据结构，所以不要指望这种方法能带来多少性能上的提高，唯一的好处就是能避免加锁带来的部分副作用。有兴趣的朋友可以看一下这篇论文，在这里我就不重复了。

# 第 5 章

# 组合的威力


5.2 栈
5.3 散列表
第5章 组合的威力
5.1 队列
第3章 从动态数组学习设计
第4章 并发与同步
第1章 从双向链表学习设计
第2章 写得又快又好的秘诀
第 0 章 背景知识

## 5.1 队列

队列是一种很常用的数据结构，操作系统用队列来管理运行的进程，驱动程序用队列来管理要传输的数据包，GUI框架用队列来管理各种GUI事件。队列是一种先进先出（FIFO，First In First Out）的数据结构，只能从队列头取数据和向队列尾追加数据。队列的名称很形象地表达了它的意义，就像排队上车一样，前面的先上，后到的站在后面排队。

队列主要的接口函数有以下这些。

- 创建队列：queue_create。
- 取队列头的元素：queue_head。
- 追加一个元素到队列尾：queue_push。
- 删除队列头元素：queue_pop。
- 取队列中元素个数（同时也可以判断队列是否为空）：queue_length。
- 遍历队列中的元素：queue_foreach。
- 销毁队列：queue_destroy。

队列看起来比链表和数组要高级一点，但其实它只不过是链表和数组的一种特殊形式而已，下面我们重用链表来实现队列。

### 队列的数据结构

```
struct _Queue
{
        DList* dlist;
};
```

这里队列只是对双向链表的一个包装。

### 创建队列

```
Queue* queue_create(DataDestroyFunc data_destroy, void* ctx)
{
        Queue* thiz = (Queue*)malloc(sizeof(Queue));

        if(thiz != NULL)
        {
            if((thiz->dlist = dlist_create(data_destroy, ctx)) == NULL)
            {
                free(thiz);
                thiz = NULL;
            }
        }
```

```
        return thiz;
}
```

创建队列时，除了分配自己的空间外，就不过是创建一个双向链表而已。

**取队列头的元素**

```
Ret queue_head(Queue* thiz, void** data)
{
        return_val_if_fail(thiz != NULL && data != NULL, RET_INVALID_PARAMS);

        return dlist_get_by_index(thiz->dlist, 0, data);
}
```

我们认为链表的第一个元素是队列头，取队列头的元素就是取链表的第一个元素。

**追加一个元素到队列尾**

```
Ret queue_push(Queue* thiz, void* data)
{
        return_val_if_fail(thiz != NULL, RET_INVALID_PARAMS);

        return dlist_append(thiz->dlist, data);
}
```



我们认为链表的最后一个元素是队列尾，追加一个元素到队列尾就是追加一个元素到链表尾。

**删除队列头元素**

```
Ret queue_pop(Queue* thiz)
{
        return_val_if_fail(thiz != NULL, RET_INVALID_PARAMS);

        return dlist_delete(thiz->dlist, 0);
}
```

删除队列头的元素就是删除链表的第一个元素。

**取队列中元素个数**

```
size_t queue_length(Queue* thiz)
{
        return_val_if_fail(thiz != NULL, 0);

        return dlist_length(thiz->dlist);
}
```

队列中元素的个数等同于链表元素的个数。

### 遍历队列中的元素

```
Ret queue_foreach(Queue* thiz, DataVisitFunc visit, void* ctx)
{
        return_val_if_fail(thiz != NULL && visit != NULL, RET_INVALID_PARAMS);

        return dlist_foreach(thiz->dlist, visit, ctx);
}
```

遍历队列中的元素等同于遍历链表中的元素。

### 销毁队列

```
void queue_destroy(Queue* thiz)
{
        if(thiz != NULL)
        {
            dlist_destroy(thiz->dlist);
            thiz->dlist = NULL;

            free(thiz);
        }

        return;
}
```

销毁链表然后释放自身的空间。

用组合的方式来实现队列很简单吧，不用半小时就可以写出来。虽然链表已经通过了自动测试，但队列的自动测试千万不能省，不过限于篇幅，在这里我们就不列出代码了。

上面我们实现的是通用队列，除了通用队列外，还有几种特殊队列应用也非常广泛，最具代表性的是以下两种。

(1) 优先级队列。它的不同之外在于，插入元素时，不必追加到队尾，而是根据元素的优先级插到队列的适当位置。这个就像排队上车时，如果后面来了个老人，大家会让他先上一样。内核的进程管理器通常采用优先级队列管理进程。

(2) 循环队列。循环队列的特点是最大元素个数是固定的。这个我们在前面学习同步时已经提过，它的优点是两个并发的实体（线程或进程）会按一个读一个写的方式访问，不需要加锁。设备驱动程序经常使用循环队列来管理数据包。

## 5.2 栈

栈是一种后进先出（LIFO，Last In First Out）的数据结构，与队列的先进先出相比，这种规则似乎不太公平，但计算机可不管这个。事实上，栈是最重要的数据结构之一。没有栈，基于下

推自动机的编译器不能工作，我们就只能写汇编程序。没有栈，无法实现递归/多级函数调用，程序根本就无法工作。

栈主要的接口函数有以下这些。

- 创建栈：stack_create。
- 取栈顶元素：stack_top。
- 放入元素到栈顶：stack_push。
- 删除栈顶元素：stack_pop。
- 取栈中元素个数：stack_length。
- 遍历栈中的元素：stack_foreach。
- 销毁栈：stack_destroy。

栈同样是链表和数组的一种特殊形式而已，下面我们重用链表来实现栈。

### 栈的数据结构

```
struct _Stack
{
        DList* dlist;
};
```

这里和队列的数据结构一样，由一个链表组成。

### 创建栈

```
Stack* stack_create(DataDestroyFunc data_destroy, void* ctx)
{
        Stack* thiz = (Stack*)malloc(sizeof(Stack));

        if(thiz != NULL)
        {
            if((thiz->dlist = dlist_create(data_destroy, ctx)) == NULL)
            {
                free(thiz);
                thiz = NULL;
            }
        }

        return thiz;
}
```

创建栈时，除了分配自己的空间外，就是简单地创建一个双向链表。

### 取栈顶元素

```
Ret stack_top(Stack* thiz, void** data)
```

```
{
        return_val_if_fail(thiz != NULL && data != NULL, RET_INVALID_PARAMS);

        return dlist_get_by_index(thiz->dlist, 0, data);
}
```

我们认为链表的第一个元素是栈顶，取栈顶的元素就是取链表的第一个元素。

**将元素放入栈顶**

```
Ret stack_push(Stack* thiz, void* data)
{
        return_val_if_fail(thiz != NULL, RET_INVALID_PARAMS);

        return dlist_prepend(thiz->dlist, data);
}
```

将元素放入栈顶就是将一个元素插入链表头。

**删除栈顶元素**

```
Ret stack_pop(Stack* thiz)
{
        return_val_if_fail(thiz != NULL, RET_INVALID_PARAMS);

        return dlist_delete(thiz->dlist, 0);
}
```

删除栈顶元素就是删除链表的第一个元素。

**获取栈中元素的个数**

```
size_t stack_length(Stack* thiz)
{
        return_val_if_fail(thiz != NULL, 0);

        return dlist_length(thiz->dlist);
}
```

栈中元素的个数等于链表中元素的个数。

**遍历栈中的元素**

```
Ret stack_foreach(Stack* thiz, DataVisitFunc visit, void* ctx)
{
        return_val_if_fail(thiz != NULL && visit != NULL, RET_INVALID_PARAMS);

        return dlist_foreach(thiz->dlist, visit, ctx);
}
```

遍历栈中的元素等同于遍历链表中的元素。

**销毁栈**

```
void stack_destroy(Stack* thiz)
{
        if(thiz != NULL)
        {
            dlist_destroy(thiz->dlist);
            thiz->dlist = NULL;

            free(thiz);
        }

        return;
}
```

销毁栈就是销毁双向链表然后释放自身的空间。

栈是一个非常重要的数据结构，但奇怪的是我们很少有机会去写它。事实上，我从来没有在工作中写过一个栈。这是怎么回事呢？原因是我们的计算机本身就是基于栈的，很多事情计算机已经在我们不知道的情况下帮我们处理了，比如函数调用（递归调用是特例），计算机都帮我们处理了。用递归下降进行的语法分析利用了函数调用的递归性，也不需要显式地构造栈。



## 5.3 散列表

前面我们已经体会到了组合的威力，用短短几十行代码就搞定了队列和栈。现在轮到散列表了，在此之前已经有几位读者向我抱怨，散列表太难写了！其实散列表也很简单，前面我们说了队列和栈只不过是链表或者数组的特殊情况而已，散列表当然不再是链表或者数组的特殊情况了，但是我们同样可以用组合的方式来实现它。简单点说就是：

**散列表 = 数组 + 链表**

有读者或许会说，老兄，你在玩我吧。不，我是认真的。我说的“加”当然不是简单地叠加起来，组合也是需要技巧的，不同的组合得到的效果不一样，如何去组合也是需要花时间去学习的。

散列表的基本接口有以下这些。

- 创建：hash_table_create。
- 插入：hash_table_insert。
- 删除：hash_table_delete。
- 查找：hash_table_find。
- 计算元素个数：hash_table_length。
- 遍历所有元素：hash_table_foreach。

- 销毁：hash_table_destroy。

现在看看怎样用数组和链表组合出散列表。

### 散列表的数据结构

```
struct _HashTable
{
        DataHashFunc     hash;
        DList**          slots;
        size_t            slot_nr;
        DataDestroyFunc  data_destroy;
        void*            data_destroy_ctx;
};
```

hash是一个函数指针，用来计算数据的散列值。散列函数的好坏基本上决定了散列表的效率，好的散列函数计算出的散列值分布比较均匀。遗憾的是散列表的设计者们谁都不知道什么样的散列函数是最好的，因为散列函数的好坏只能动态评估，它与数据类型和应用环境密切相关。按照惯例，实现者不知道的事就应该让调用者去实现，所以把散列函数设计成回调函数，由调用者提供。

slots是散列表的主体，它是一个双向链表的指针数组。所以说“散列表＝数组＋链表”是有道理的。由于这个数组不需要动态增长，所以用最简单的指针数组就好了。

slot_nr是数组的大小，在散列函数不变的情况下，slot_nr的大小对散列表的性能起决定作用。slot_nr的值越大性能越高，但空间浪费也越大，这又是一个时/空互换的例子。所以这个值也由调用者确定会好一点。很多书都认为这个值应该选择一个素数，我认为这种做法没有什么科学的理论根据，至少没有找到严密的数学证明。

### 创建散列表（hash_table_create）

```
HashTable* hash_table_create(DataDestroyFunc data_destroy, void* ctx, DataHashFunc
hash, int slot_nr)
{
        HashTable* thiz = NULL;

        return_val_if_fail(hash != NULL && slot_nr > 1, NULL);

        thiz = (HashTable*)malloc(sizeof(HashTable));

        if(thiz != NULL)
        {
            thiz->hash = hash;
            thiz->slot_nr  = slot_nr;
            thiz->data_destroy_ctx = ctx;
            thiz->data_destroy = data_destroy;
            if((thiz->slots = (DList**)calloc(sizeof(DList*)*slot_nr, 1)) == NULL)
```

```
        {
            free(thiz);
            thiz = NULL;
        }
    }

    return thiz;
}
```

创建散列表时，我们只是创建了数组，而链表则在第一次使用时再创建。这种延迟处理的手法在加快启动速度时是很常见的，这种做法也会减少一些不必要的开销（因为有些对象可能根本就不会用到）。

**插入散列表（hash_table_insert）**

```
Ret hash_table_insert(HashTable* thiz, void* data)
{
    size_t index = 0;

    return_val_if_fail(thiz != NULL, RET_INVALID_PARAMS);
  /*先通过hash值定位到链表。*/
    index = thiz->hash(data)%thiz->slot_nr;
    if(thiz->slots[index] == NULL)
  {
        /*链表不存在就创建它。*/
        thiz->slots[index] = dlist_create(thiz->data_destroy
    , thiz->data_destroy_ctx);
    }
    /*增加数据到链表中*/
    return dlist_prepend(thiz->slots[index], data);
}
```

先计算元素所在链表，如果链表还没有创建，我们就创建它，然后把元素插入到链表中。怎样？也不过是几行代码而已。

**删除散列表（hash_table_delete）**

```
Ret hash_table_delete(HashTable* thiz, DataCompareFunc cmp, void* data)
{
    int index = 0;
    DList* dlist = NULL;

    return_val_if_fail(thiz != NULL && cmp != NULL, RET_INVALID_PARAMS);
    /*先通过hash值定位到链表。*/
    index = thiz->hash(data)%thiz->slot_nr;
    dlist = thiz->slots[index];
    if(dlist != NULL)
  {
     /*删除数据*/
        index = dlist_find(dlist, cmp, data);
```

```
            return dlist_delete(dlist, index);
        }

        return RET_FAIL;
}
```

先计算元素所在的链表，然后从链表中删除元素。

### 查找散列表的元素（hash_table_find）

```
Ret hash_table_find(HashTable* thiz, DataCompareFunc cmp, void* data,
void** ret_data)
{
        int index = 0;
        DList* dlist = NULL;
        return_val_if_fail(thiz != NULL && cmp != NULL && ret_data != NULL,
    RET_INVALID_PARAMS);
        /*先通过hash值定位到链表。*/
        index = thiz->hash(data)%thiz->slot_nr;
        dlist = thiz->slots[index];
        if(dlist != NULL)
        {
            index = dlist_find(dlist, cmp, data);

            return dlist_get_by_index(dlist, index, ret_data);
        }

        return RET_FAIL;
}
```

先计算元素所在的链表，然后从链表中查找元素。

### 计算元素个数（hash_table_length）

```
size_t  hash_table_length(HashTable* thiz)
{
        size_t i = 0;
        size_t nr = 0;

        return_val_if_fail(thiz != NULL, 0);

        /*累加所有链表中元素的个数*/
    for(i = 0; i < thiz->slot_nr; i++)
        {
            if(thiz->slots[i] != NULL)
            {
                nr += dlist_length(thiz->slots[i]);
            }
        }

        return nr;
}
```

这个麻烦一点，需要累加所有链表中元素个数。

### 遍历所有元素（hash_table_foreach）

```
Ret hash_table_foreach(HashTable* thiz, DataVisitFunc visit, void* ctx)
{
        size_t i = 0;

        return_val_if_fail(thiz != NULL && visit != NULL, RET_INVALID_PARAMS);

        for(i = 0; i < thiz->slot_nr; i++)
        {
            if(thiz->slots[i] != NULL)
            {
                dlist_foreach(thiz->slots[i], visit, ctx);
            }
        }

        return RET_OK;
}
```

依次调用每个链表的dlist_foreach。

### 销毁散列表（hash_table_destroy）

```
void hash_table_destroy(HashTable* thiz)
{
        size_t i = 0;

        if(thiz != NULL)
        {
            for(i = 0; i < thiz->slot_nr; i++)
            {
                if(thiz->slots[i] != NULL)
                {
                dlist_destroy(thiz->slots[i]);
                thiz->slots[i] = NULL;
                }
            }

            free(thiz->slots);
            free(thiz);
        }

        return;
}
```

销毁所有链表，释放数组和散列表本身。

散列表有以下两种特殊情况。

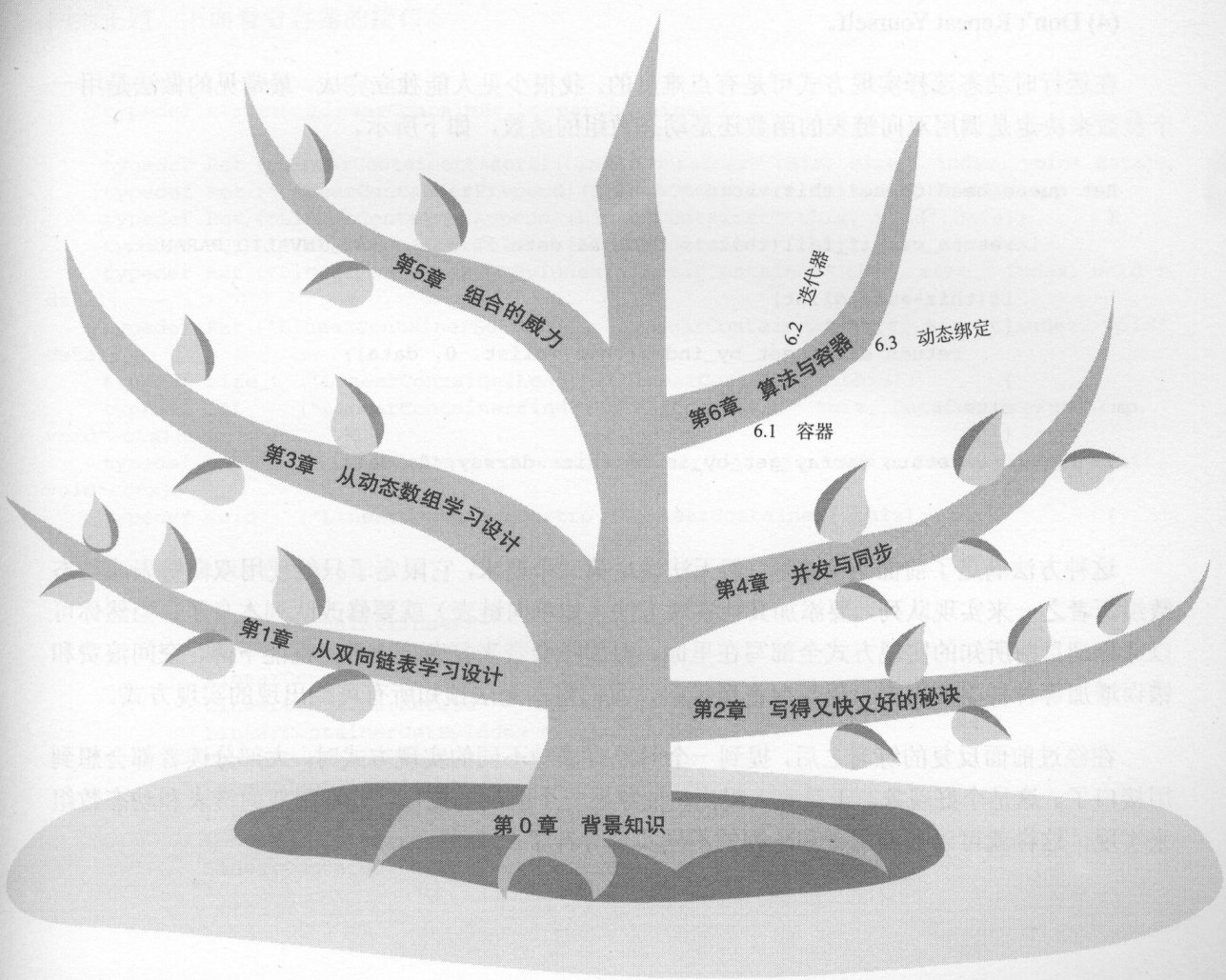
(1) 散列函数极差：所有元素计算出同一个散列值。则散列表退化成一个链表，查找的时间效率为O($n$)。

(2) 散列函数极好：所有元素计算出不同的散列值，而且元素个数为slot_nr。则散列表等同于一个数组，通过索引定位元素，查找的时间效率为O(1)。

所以散列表的查找效率在O(1)到O($n$)之间，大部分人选择散列表时，并没有仔细评估散列函数的好坏，而又期望得到很高的查找效率，其实这只是一种想当然的看法而已。如果查找效率要求比较高，通常我会选择有序数组，用二分查找来做，至少它的查找效率是比较确定的。

# 第 6 章

# 算法与容器


第5章　组合的威力
第6章　算法与容器
6.1　容器
6.2　迭代器
6.3　动态绑定
第3章　从动态数组学习设计
第4章　并发与同步
第1章　从双向链表学习设计
第2章　写得又快又好的秘诀
第 0 章　背景知识

## 6.1 容器

前面我们通过组合的方法，用双向链表实现了队列。大家已经看到，它的实现非常简单。有的读者可能会有疑问，你说过双向链表和动态数组都有各自的适用范围，在有的情况下，用双向链表合适，而在有的情况下，用动态数组合适，那么你用双向链表实现了队列，还有必要用动态数组再实现一次吗？如果对栈同样如此，那是不是做了太多重复的工作，违背了DRY（don't repeat yourself）原则呢？

问得好，这就是本章要学习的。我们请读者实现一个队列，要求如下。

(1) 由调用者决定用双向链表实现还是用动态数组实现。
(2) 在运行时决定，而不是在编译时决定（宏定义）。
(3) 如果调用者乐意，他以后还可以选择用单向链表来实现，而不必修改队列的实现代码。
(4) Don't Repeat Yourself。

在运行时动态选择实现方式可是有点难度的。我很少见人能独立完成，最常见的做法是用一个参数来决定是调用双向链表的函数还是动态数组的函数，如下所示。

```
Ret queue_head(Queue* thiz, void** data)
{
        return_val_if_fail(thiz != NULL && data != NULL, RET_INVALID_PARAMS);

        if(thiz->use_dlist)
        {
            return dlist_get_by_index(thiz->dlist, 0, data);
        }
        else
        {
            return darray_get_by_index(thiz->darray, 0, data);
        }
}
```

这种方法满足了前面两个要求，但无法满足第三个要求，它限定了只能使用双向链表或动态数组两者之一来实现队列，要添加其他实现方法（如单向链表）就要修改队列本身了。当然你可以选择把目前所知的实现方式全部写在里面，但那样会带来复杂度上升、性能下降、空间浪费和错误增加等种种不利因素。更何况在现实中，我们根本无法预知所有可能出现的实现方式。

在经过前面反复的练习之后，提到一个对象有多种不同的实现方式时，大部分读者都会想到用接口了。这是个好现象。于是有人提出，抽象出一个队列的接口，分别用双向链表和动态数组来实现，这样就可以把调用者和队列的不同实现分开了。

这个想法还不错，至少能通过队列接口解决前三个问题了。但问题是，用双向链表实现的队列和用动态数组实现的队列，从逻辑上来看，完全是重复的，从代码的角度来看，它们又有些差别。如果你亲手去写过，总感觉到里面有点不太对劲，这种感觉通常是需要改进的征兆。在仔细思考之后，我们会发现队列的逻辑是不变的，变化的只是容器。也就是说应该抽象的是容器接口而不是队列接口。

这里我还要讲讲“最小粒度抽象原则”。这个名称是我起的，虽然不是大师之言，但它同样是有效的。接口隔离了调用者和具体实现，不管抽象的粒度大小如何，接口都能起到隔离变化的作用。但是粒度越大，造成的重复就越多。上面的例子或许不太明显，但假设我们要开发一个OS内核，这个OS可以在不同的硬件平台上运行，硬件平台是变化的，我们只要对硬件平台进行抽象就好了，如果对OS整体进行抽象，则意味着每个硬件平台写一个OS，那样就存在太多重复工作了。

前面我们在引入Locker这个接口时，是先有这个接口然后再去实现它。这里不过是已经有实现好的双向链表和动态数组，我们再引入容器这个接口而已。它们的目的是一样的：隔离变化。回到正题，下面看看容器的接口。

```
    struct _LinearContainer;
    typedef struct _LinearContainer LinearContainer;

    typedef Ret (*LinearContainerInsert)(LinearContainer* thiz, size_t index, void* data);
    typedef Ret (*LinearContainerPrepend)(LinearContainer* thiz, void* data);
    typedef Ret (*LinearContainerAppend)(LinearContainer* thiz, void* data);
    typedef Ret (*LinearContainerDelete)(LinearContainer* thiz, size_t index);
    typedef Ret (*LinearContainerGetByIndex)(LinearContainer* thiz, size_t index, void**
data);
    typedef Ret (*LinearContainerSetByIndex)(LinearContainer* thiz, size_t index, void*
data);
    typedef size_t (*LinearContainerLength)(LinearContainer* thiz);
    typedef int    (*LinearContainerFind)(LinearContainer* thiz, DataCompareFunc cmp,
void* ctx);
    typedef Ret    (*LinearContainerForeach)(LinearContainer* thiz, DataVisitFunc visit,
void* ctx);
    typedef void   (*LinearContainerDestroy)(LinearContainer* thiz);

    struct _LinearContainer
    {
        LinearContainerInsert     insert;
        LinearContainerPrepend    prepend;
        LinearContainerAppend     append;
        LinearContainerDelete     del;
        LinearContainerGetByIndex get_by_index;
        LinearContainerSetByIndex set_by_index;
        LinearContainerLength     length;
        LinearContainerFind       find;
        LinearContainerForeach    foreach;
        LinearContainerDestroy    destroy;
```

```
        char priv[0];
};
```

从上面的代码可以看出，容器接口、双向链表和动态数组的实现基本上是一致的。

下面我们看看基于双向链表的实现。

**数据结构**

```
typedef struct _PrivInfo
{
        DList* dlist;
}PrivInfo;
```

这是容器的私有数据，它只是对双向链表所进行的包装，请不要与前面的“组合”搞混了：组合是为了实现新的功能，而这里是为了分隔接口和实现，让实现部分可以独立变化。

**实现接口函数**

```
static Ret linear_container_dlist_insert(LinearContainer* thiz, size_t index, void* data)
{
        PrivInfo* priv = (PrivInfo*)thiz->priv;

        return dlist_insert(priv->dlist, index, data);
}

static Ret linear_container_dlist_delete(LinearContainer* thiz, size_t index)
{
        PrivInfo* priv = (PrivInfo*)thiz->priv;

        return dlist_delete(priv->dlist, index);
}

...
```

同样这里只是对参数做简单转发，调用都是一一对应的，其他函数的实现类似，这里就不一个一个地分析了。

**创建函数**

```
LinearContainer* linear_container_dlist_create(DataDestroyFunc data_destroy,
void* ctx)
{
        LinearContainer* thiz = (LinearContainer*)malloc(sizeof(LinearContainer) +
    sizeof(PrivInfo));

        if(thiz != NULL)
        {
            PrivInfo* priv = (PrivInfo*)thiz->priv;
            priv->dlist = dlist_create(data_destroy, ctx);
```

```
            thiz->insert        =  linear_container_dlist_insert;
            thiz->prepend       =  linear_container_dlist_prepend;
            thiz->append        =  linear_container_dlist_append;
            thiz->del           =  linear_container_dlist_delete;
            thiz->get_by_index  =  linear_container_dlist_get_by_index;
            thiz->set_by_index  =  linear_container_dlist_set_by_index;
            thiz->length        =  linear_container_dlist_length;
            thiz->find          =  linear_container_dlist_find;
            thiz->foreach       =  linear_container_dlist_foreach;
            thiz->destroy       =  linear_container_dlist_destroy;

            if(priv->dlist == NULL)
            {
                free(thiz);
                thiz = NULL;
            }
        }
    }
```

创建函数的主要工作是把函数指针指向实际的函数。

看到这里，有的读者可能会问，你用双向链表和动态数组实现了两个容器，我用双向链表和动态数组实现了两个队列，这没有什么差别啊，至少你的代码量并不会减少。能想到这点是不错的，所谓不怕不识货就怕货比货，这两种方法到底有什么差别呢？

前面我们说过，在元素很少的情况下，快速排序的性能甚至不如冒泡排序高，但是随着元素个数的增加，它的优势就越来越明显了。其实好的设计也是一样的，在小程序中，它的表现并不见得比普通设计强多少，但是随着规模的扩大，重用次数的增多，它的优势就会明显起来。单从实现队列来看，容器接口没有什么优势。眼光再看远一点，我们会发现：用同样的方法实现栈不需要重写这些代码了，双向链表和动态数组可以共用一个测试程序。如果其他地方还有类似的需求，重用次数会越来越多，它的优势就更明显了。

下面我们来看看用容器实现队列。

**数据结构**

```
struct _Queue
{
        LinearContainer* linear_container;
};
```

**队列的创建函数**

```
Queue* queue_create(LinearContainer* container)
{
        Queue* thiz = NULL;
```

```
    return_val_if_fail(container != NULL, NULL);

    thiz = (Queue*)malloc(sizeof(Queue));

    if(thiz != NULL)
    {
        thiz->linear_container = container;
    }

    return thiz;
}
```

容器对象由调用者传入，而不是在队列里硬编码，这样它的变化不会影响队列。

**队列的实现函数**

```
Ret queue_head(Queue* thiz, void** data)
{
    return_val_if_fail(thiz != NULL && data != NULL, RET_INVALID_PARAMS);

    return linear_container_get_by_index(thiz->linear_container, 0, data);
}

Ret queue_push(Queue* thiz, void* data)
{
    return_val_if_fail(thiz != NULL, RET_INVALID_PARAMS);

    return linear_container_append(thiz->linear_container, data);
}
...
```

用容器实现的队列和前面用双向链表实现的队列没有多大差别，只是函数名称不一样而已。

由此可见，队列本身与双向链表和动态数组是没有任何关系的，队列关心的只是linear_container这个接口而已，如果调用者想换用单向链表来实现队列，只要用单向链表去实现linear_container接口就好了。对使用C++的读者而言，在C++的标准模板库（STL）里也有类似的做法，STL里的Sequence就相当于这里的LinearContainer接口。

C语言程序中广泛使用上述做法，但我并不推荐（当然也不反对）读者这样去实现队列。原因是在通常情况下，队列中存放的元素并不多，使用双向链表、动态数组和其他容器都没有多大差别，其实只要用最简单的方法实现它就行了。

## 6.2 迭代器

容器用来存储数据，算法用来处理数据。容器有多种，算法的种类更多，两者的组合数目就数不胜数了。如果同样的算法要为每种容器都写一遍，写的时候单调不说，维护起来也很困难。所

以我们一直在寻找让算法独立于容器的方法，让同一种算法能适用于所有（或尽可能多）的容器。

在开始的时候，我们通过foreach遍历函数和回调函数，第一次分离了算法和容器，算法不但可以独立于容器变化，而且同时适用于多种容器：大/小写转换函数只需要写一次，就可以在双向链表、动态数组、队列、栈和散列表等多种容器中重用。

前面又我们抽象了容器的接口，容器的使用者不需要关心容器的实现，也就是说同一个算法可以适用于多种不同的容器（不是全部，前面只抽象了线性的容器）。我们用这种方面实现了队列和栈，还重用了双向链表和动态数组的测试程序。

可惜上面两种方法仍然不能解决让算法独立于容器的全部问题。这里请读者先实现这样一个算法。

(1) 反序排列容器中的元素：第一个元素与最后一个元素交换，第二个元素与倒数第二个元素交换，以此类推。

(2) 这个算法同时适用于双向链表和动态数组。

(3) 在双向链表和动态数组上的运行效率处于同一级别。

这个反序函数的原理很简单，有的读者应该很快就写出来了。

```
Ret invert(LinearContainer* linear_container)
{
        int i = 0;
        int j = 0;
        void* data1 = NULL;
        void* data2 = NULL;

        return_val_if_fail(linear_container != NULL, RET_INVALID_PARAMS);

        j = linear_container_length(linear_container) - 1;
        for(; i < j; i++, j--)
        {
            linear_container_get_by_index(linear_container, i, &data1);
            linear_container_get_by_index(linear_container, j, &data2);
            linear_container_set_by_index(linear_container, i, data2);
            linear_container_set_by_index(linear_container, j, data1);
        }

        return RET_OK;
}
```

这种实现利用我们抽象的 LinearContainer接口，它同时适用于双向链表和动态数组，可惜无法满足第三个条件，双向链表和动态数组的性能差异很大。原因是动态数组通过偏移量可以定位，而双向链表则需要一个一个地移动指针才能定位，在循环内部调用get_by_index/set_by_index进一步加剧了它们在性能上的差异。

有的读者可能尝试过用foreach去实现，foreach确实很好用，但它的缺点是只能按固定的顺序去遍历元素，不能同时既反序又顺序地遍历，所以用foreach实现上述算法是有点困难的。这里我们引入一个新的概念：迭代器。迭代器是一种更灵活的遍历行为的抽象，它可以按任意顺序去访问容器内的元素，而且不会暴露容器的内部结构。

在引入迭代器之前，我们分析一下invert的算法，先考虑针对双向链表的实现。

(1) 定义两个指针，一个指向链表的首结点，一个指向链表的尾结点。
(2) 交换两个指针指向的内容。
(3) 前面的指针向后移，后面的指针向前移，重复第二步直到两个指针相遇。

仔细看看上面的算法，我们发现在整个过程中，操作的只是两个指针。关于这个指针，它具有下列行为：

(1) 获取指针指向的数据；
(2) 设置指针指向的数据；
(3) 指针向后移；
(4) 指针向前移；
(5) 比较大小。

在前面的算法中，指针是依赖于双向链表的，离开了这个特定的容器，我们就无法确定指针的行为。那么我们可以对这里的指针进行抽象，让它针对不同容器有不同的实现，而算法只关心它的指针这个接口。为了遵循一些约定俗成的规则，我们把这里的指针称为迭代器（iterator）。

迭代器的接口定义如下：

```
typedef Ret  (*IteratorSetFunc)(Iterator* thiz, void* data);
typedef Ret  (*IteratorGetFunc)(Iterator* thiz, void** data);
typedef Ret  (*IteratorNextFunc)(Iterator* thiz);
typedef Ret  (*IteratorPrevFunc)(Iterator* thiz);
typedef Ret  (*IteratorAdvanceFunc)(Iterator* thiz, int offset);
typedef int  (*IteratorOffsetFunc)(Iterator* thiz);
typedef Ret  (*IteratorCloneFunc)(Iterator* thiz, Iterator** cloned);
typedef void (*IteratorDestroyFunc)(Iterator* thiz);

struct _Iterator
{
        IteratorSetFunc     set;
        IteratorGetFunc     get;
        IteratorNextFunc    next;
        IteratorPrevFunc    prev;
        IteratorAdvanceFunc advance;
        IteratorCloneFunc   clone;
        IteratorOffsetFunc  offset;
```

```
        IteratorDestroyFunc destroy;

        char priv[0];
};
```

为了让迭代器也可以在其他算法中使用，这里增加了几个接口函数。

(1) advance：随机定位迭代器位置，由当前位置和偏移量决定。

(2) offset：得到迭代器的偏移量，主要用于比较迭代器的大小。

(3) clone：复制一份迭代器。

现在我们来看用双向链表实现的迭代器。

**数据结构**

```
typedef struct _PrivInfo
{
    DList* dlist;
    DListNode* cursor;
    int offset;
}PrivInfo;
```

它包括一个指向双向链表的指针，它表明迭代器所属的容器；一个指向当前结点的指针，它表明迭代器的位置；一个标明当前偏移量的值，目的是提高获取偏移量的速度。



**实现迭代器的函数**

```
static Ret  dlist_iterator_set(Iterator* thiz, void* data)
{
        PrivInfo* priv = (PrivInfo*)thiz->priv;
        return_val_if_fail(priv->cursor != NULL && priv->dlist != NULL,
    RET_INVALID_PARAMS);

        priv->cursor->data = data;

        return RET_OK;
}

static Ret  dlist_iterator_next(Iterator* thiz)
{
        Ret ret = RET_FAIL;
        PrivInfo* priv = (PrivInfo*)thiz->priv;
        return_val_if_fail(priv->cursor != NULL && priv->dlist != NULL,
    RET_INVALID_PARAMS);

        if(priv->cursor->next != NULL)
        {
            priv->cursor = priv->cursor->next;
            priv->offset++;
```

```
            ret = RET_OK;
        }

        return ret;
}

...
```

从这些代码可以看出，双向链表的迭代器其实就是对双向链表结点的包装，经过这个包装之后，访问具体的结点不会暴露双向链表的内部实现，而且调用者可以不再依赖双向链表了。

**创建函数**

```
Iterator* dlist_iterator_create(DList* dlist)
{
        Iterator* thiz = NULL;
        return_val_if_fail(dlist != NULL, NULL);

        if((thiz = malloc(sizeof(Iterator) + sizeof(PrivInfo))) != NULL)
        {
            PrivInfo* priv = (PrivInfo*)thiz->priv;

            thiz->set       =  dlist_iterator_set;
            thiz->get       =  dlist_iterator_get;
            thiz->next      =  dlist_iterator_next;
            thiz->prev      =  dlist_iterator_prev;
            thiz->advance   =  dlist_iterator_advance;
            thiz->clone     =  dlist_iterator_clone;
            thiz->offset    =  dlist_iterator_offset;
            thiz->destroy   =  dlist_iterator_destroy;

            priv->dlist  = dlist;
            priv->cursor = dlist->first;
            priv->offset = 0;
        }

        return thiz;
}
```

这里我们还遇到另外一个难题：双向链表的迭代器的实现放在哪里？放在dlist.c里还是单独的文件里呢，放在单独的文件里可读性更强，但是在访问双向链表的内部数据时会遇到一些麻烦。最后的选择如下。

(1) dlist_iterator.h提供独立的文件。

(2) dlist_iterator.c也提供独立的文件，但是它不直接参与编译，而是在dlist.c里include它。

这样就两全其美了：维护方便又可以访问双向链表的内部数据结构。

最后我们来看看用迭代器实现的invert函数。

```
Ret invert(Iterator* forward, Iterator* backward)
{
        void* data1 = NULL;
        void* data2 = NULL;
        return_val_if_fail(forward != NULL && backward != NULL, RET_INVALID_PARAMS);

        for(; iterator_offset(forward) < iterator_offset(backward);
    iterator_next(forward), iterator_prev(backward))
        {
            iterator_get(forward, &data1);
            iterator_get(backward, &data2);
            iterator_set(forward, data2);
            iterator_set(backward, data1);
        }

        return RET_OK;
}
```

invert算法同时适用于双向链表和动态数组，而且它们的运行效率相差不大，至少对于双向链表来说已经是比较高效的实现了。

迭代器模式是一种重要的模式，在C++的标准模板库（STL）中有大量的应用。老的C程序员通常是运行效率至上的，所以要在优雅的设计和高性能之间做出选择时，他们通常选择后者，所以很少看到C语言实现的容器提供迭代器，那也不足为奇。重要的是我们可以学习这种方法，并在适当的时候运用它。



## 6.3 动态绑定

前面我们通过容器接口抽象了双向链表和动态数组，这样队列的实现就不依赖于具体的容器了。但是作为队列的使用者，它仍然要在编译时决定使用哪个容器。队列的测试程序就是队列的使用者之一，它的实现代码如下。

```
Queue* queue = queue_create(linear_container_dlist_create(NULL, NULL));

    for(i = 0; i < n; i++)
    {
        assert(queue_push(queue, (void*)i) == RET_OK);
        assert(queue_head(queue, (void**)&ret_data) == RET_OK);
        assert(queue_length(queue) == (i+1));
    }
  ...
```

是linear_container_dlist_create还是linear_container_darray_create，这里需要我们明确地指定。而假设使用者想要换一种容器，那还是要修改代码并重新编译才行。现在我们思考另外一个问题：如何让使用者（如这里的测试程序）在想换用另一种容器时，既不需要修改

代码也不需要重新编译。

在继续学习之前，我们先介绍几个概念。

- 静态库：在Linux下，静态库的扩展名为.a，a代表archive（档案、存档）的意思。正常情况下一个C源文件编译之后生成一个目标文件（.o），目标文件里存放的是程序的机器指令和数据，它不包含运行时的信息，所以不能直接运行。用命令ar把多个目标文件打包成一个文件就生成了所谓的静态库。可执行文件链接静态库时，会把用到的函数和数据复制过去。多个可执行文件链接同一个静态库时，所用到的函数和数据就会被复制多次，那就存在不少空间浪费。
- 共享库：顾名思义，共享库可以在多个可执行文件之间共享，链接时不用拷贝函数或数据，只是建立一个函数链接表（PLT），在运行时通过这个表来确定具体调用的函数。共享库可以有效地避免空间浪费，但它也不是免费的午餐，它在加载时存在额外的开销，在链接多个共享库时会比较明显。不过目前出现的prelink和gnu hash等技术，有效地缓解了这个问题。共享库另外一个好处就是可以单独升级，理论上，修改某个共享库不需要重新编译依赖它的应用程序（不过实际操作时会与它的接口变化和信息隐藏程度有关）。
- 可执行文件：Linux下加x属性的文件都是可执行文件。这里所说的“可执行文件”特指ELF（Executable and Linking Format）格式的可执行文件。可执行文件在编译时连接静态库，所用到的函数会被复制过来，运行时不再依赖于静态库。在编译时链接共享库，它不拷贝函数和数据，但在运行时会依赖于其链接的共享库。

回到前面的问题。我们发现在编译时，无论是链接静态库还是共享库，都绑定了调用者与使用者之间的关系。真正要在运行时决定而不是编译时决定使用哪种容器，那就不能包含具体容器的头文件，链接具体容器所在的库了。今天我们要学习一种新的技术：在运行时动态加载共享库。

除了一些嵌入式环境下使用的实时操作系统（RTOS）外，现代操作系统都支持在运行时动态加载共享库的机制。Linux下有dlopen系列函数，Windows下有LoadLibrary系列函数，其他平台也有类似的函数。下面是使用dlopen的简单示例。

```
#include <stdio.h>
#include <stdlib.h>
#include <dlfcn.h>

int main(int argc, char **argv)
{
        void *handle = NULL;
        double (*cosine)(double) = NULL;
        /*加载共享库*/
        handle = dlopen("libm.so", RTLD_LAZY);
        /*通过函数名找到函数指针*/
        *(void **) (&cosine) = dlsym(handle, "cos");
        /*调用函数*/
```

```
        printf("%f\n", (*cosine)(2.0));
        /*卸载共享库*/
        dlclose(handle);

        return 0;
}
```

由于这些函数在每个平台的名称和参数都有所不同，直接使用这些函数会带来可移植性的问题，为此有必要对它们进行包装。不同平台有不同的函数，也就是说存在多种不同的实现，那这是否意味着要用接口呢？答案是不用。原因是同一个平台只有一种实现，而且不会出现潜在的变化。我们要做的只是添加一个适配层，用它来隔离不同平台就好了。

我们把这个加载函数的适配层称为Module，声明（module.h）如下。

```
struct _Module;
typedef struct _Module Module;

typedef enum _ModuleFlags
{
        MODULE_FLAGS_NONE,
        MODULE_FLAGS_DELAY = 1
}ModuleFlags;

Module* module_create(const char* file_name, ModuleFlags flags);
void*   module_sym(Module* thiz, const char* func_name);
void    module_destroy(Module* thiz);
```



这里它们的实现只是对dl系列函数做个简单包装。由于不同平台有不同的实现，为了维护方便，我们把不同的实现放在不同的文件中，比如Linux的实现放在module_linux.c里。

```
Module* module_create(const char* file_name, ModuleFlags flags)
{
        Module* thiz = NULL;
        return_val_if_fail(file_name != NULL, NULL);

        if((thiz = malloc(sizeof(Module))) != NULL)
        {
            thiz->handle = dlopen(file_name,
                    flags & MODULE_FLAGS_DELAY ? RTLD_LAZY : RTLD_NOW);
            if(thiz->handle == NULL)
            {
                free(thiz);
                thiz = NULL;
                printf("%s\n", dlerror());
            }
        }

        return thiz;
}
```

我们再看看队列的测试程序要怎么写。

```
#include "linear_container.h"

typedef LinearContainer* (*LinearContainerDarrayCreateFunc)(DataDestroyFunc data_
destroy, void* ctx);

int main(int argc, char* argv[])
{
        int i = 0;
        int n = 1000;
        int ret_data = 0;
        Queue* queue = NULL;
        Module* module = NULL;
        LinearContainerDarrayCreateFunc linear_container_create = NULL;
        if(argc != 3)
        {
            printf("%s sharelib linear_container_create\n", argv[0]);

            return 0;
        }

        module = module_create(argv[1], 0);
        return_val_if_fail(module != NULL, 0);
        linear_container_create = (LinearContainerDarrayCreateFunc)module_sym(module,
argv[2]);
        return_val_if_fail(linear_container_create != NULL, 0);

        queue = queue_create(linear_container_create(NULL, NULL));
        ...
}
```

这里有两点需要说明。

(1) 头文件只包含linear_container.h，而不包含linear_container_dlist.h。

(2) 我们要通过module_sym获取容器的创建函数，而不要直接调用linear_container_dlist_create。

在这里，编译时可以不再链接容器所在的共享库。

```
gcc -Wall -g module_linux.c queue.c queue_test.c -ldl -o queue_dynamic_test
```

通过以下代码运行自己决定要使用的容器（这里是采用双向链表）。

```
./queue_dynamic_test ./libcontainer.so linear_container_dlist_create
```

这样一来，容器的调用者和使用者便完全分隔开了。“接口＋动态加载”的方式也称为插件式设计，它是软件可扩展性的基础，像操作系统、办公软件、浏览器和Web服务器等大型软件都无一例外地使用了这类技术。

到第6章为止本书的上半部分已经完成了。在上半部分中，我们学习了基本的数据结构、算法和设计思想。在进行深入学习之前，我们把前面所写的代码整理成一个通用的函数库，这个函

数库可能在以后的工作中用得着。

前面我们写的Makefile非常简单，大概类似于如下所示。

```
all:
        gcc -Wall -g -DDARRAY_TEST darray.c -o darray_test
        gcc -Wall -g -DDLIST_TEST dlist.c -o dlist_test
        gcc -Wall -g linear_container_test.c -L./ -lcontainer -o container_test
        gcc -Wall -g invert_ng.c -DINVERT_TEST -L./ -lcontainer -o invert_ng_test
        gcc -Wall -g invert.c -DINVERT_TEST -L./ -lcontainer -o invert_test
        gcc -Wall -g -shared darray.c  dlist.c  linear_container_darray.c
    linear_container_dlist.c -o libcontainer.so
clean:
        rm *test *.so
```

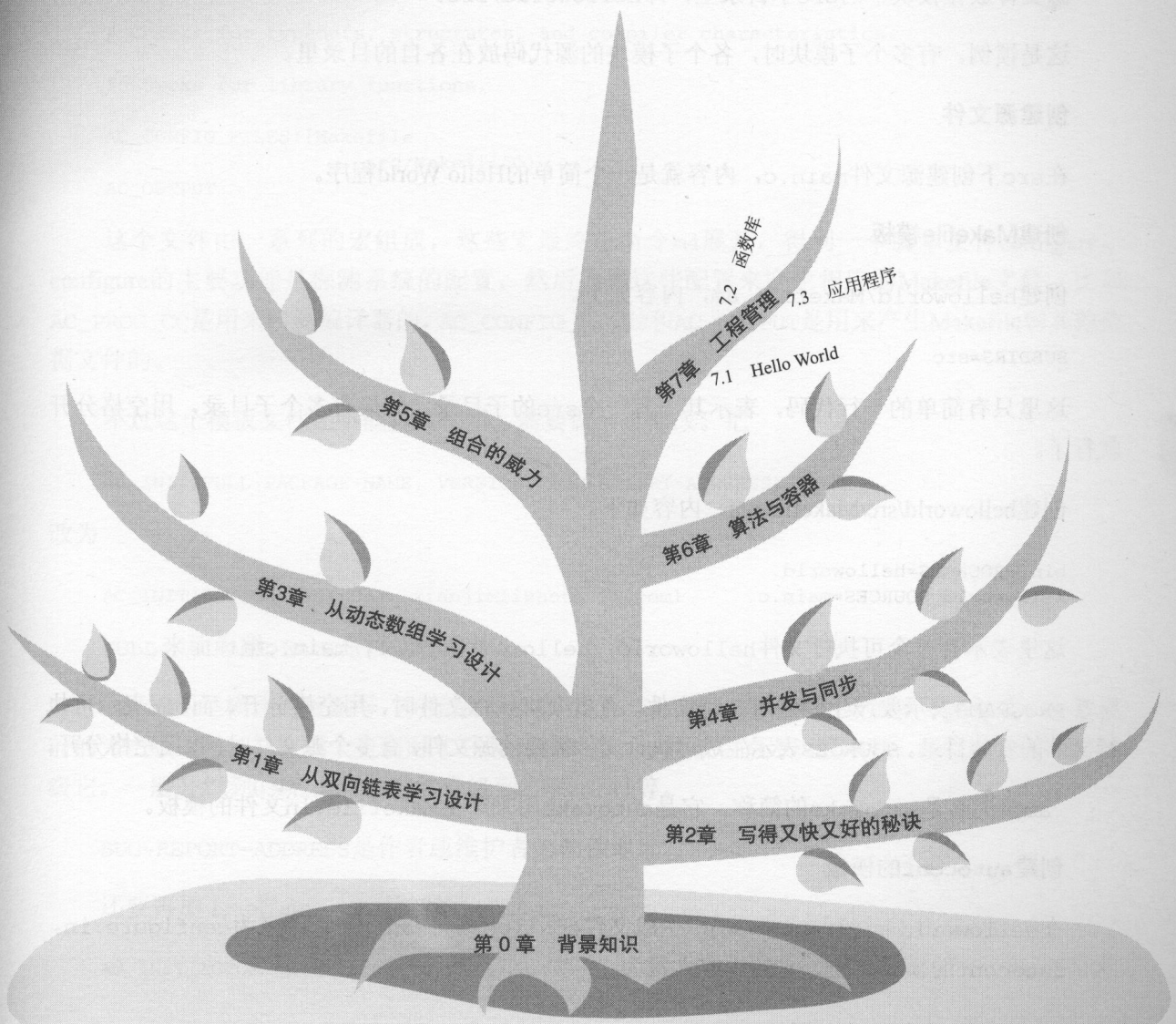
这个简单的Makefile对于我们学习编程已经够了，因为我们写的函数库没有真正的用户，只要测试程序通过就好了。但是作为一个真正的软件包，你还要考虑下列因素。

- 依赖规则。前面我们没有写编译依赖规则，不管源文件有没有变化，反正全部编译一遍。对于小程序来说用不了多少时间，那没有问题，而一些大模块可能要花上几十分钟，那就不可接受了。
- 不同平台之间的差异。即使同是支持POSIX标准的类Unix操作系统，也多多少少都会有些不同。同样是Linux，发行版本不一样、libc（除了glibc外还有几种轻量级libc）不一样，或者使用的桌面环境（如KDE和GNOME）不一样，都会给编写跨平台软件来带一些困难。我们在编写软件时已经考虑到了可移植性问题，但是去检查编译环境却是件麻烦的事。
- 交叉编译。我们在PC上编译在嵌入式设备上运行的程序，或者在x86架构上编译在PowerPC架构上运行的程序，这都是属于交叉编译。交叉编译产生不同于编译过程所在机器的机器码，这要求编译时使用不同（版本）的编译器。
- 同时为多个平台编译。比如编译一个在PC上运行的模拟环境和在ARM板子上实际运行的版本。在本书前面所写的Makefile里，编译生成的文件会全部放在一个目录下，一次只能编译一个版本，而且在切换到另外一个版本时，你要清除所有文件重新编译。
- 调用者如何使用。一个软件包肯定不是孤立存在的，一定有另外的函数库或应用程序使用它。它们有的安装在/usr下，有的安装在/usr/local下，还有的安装到用户目录下。调用者怎么知道这些软件包安装在哪里呢？不知道安装在哪里，又怎样去链接它呢？
- Makefile的复杂度。要用Makefile解决上述问题，Makefile一定变得非常复杂。有兴趣的读者可以研究一下Firefox、Linux和Android的Makefile，这些Makefile可不是一般人可以写出来的，普通开发人员通常不愿意花几周时间去写Makefile吧。

其实这些问题都不用担心，因为开源界的前辈们早就遇到过了，并用非常巧妙的方式解决了它们。接下来，我们一起学习automake/autoconf，看看它是如何解决这些问题的。

# 第 7 章

# 工程管理


第7章 工程管理
7.1 Hello World
7.2 函数库
7.3 应用程序
第6章 算法与容器
第5章 组合的威力
第4章 并发与同步
第3章 从动态数组学习设计
第2章 写得又快又好的秘诀
第1章 从双向链表学习设计
第 0 章 背景知识

## 7.1 Hello World

automake比起IDE要复杂很多，这里我们先写一个Hello World例子，明白其中的基本概念后，再用它来管理实际的工程。

### 目录结构

最顶层目录名用模块名称，这里是helloworld。

源文件放在模块下的src子目录里，即helloworld/src。

这是惯例。有多个子模块时，各个子模块的源代码放在各自的目录里。

### 创建源文件

在src下创建源文件main.c，内容就是一个简单的Hello World程序。

### 创建Makefile模板

创建helloworld/Makefile.am，内容如下。

```
SUBDIRS=src
```

这里只有简单的一行代码，表示其下有一个src的子目录，如果有多个子目录，用空格分开就行了。

创建helloworld/src/Makefile.am，内容如下。

```
bin_PROGRAMS=helloworld
helloworld_SOURCES=main.c
```

这里表示有一个可执行文件helloworld，helloworld由源文件main.c编译而来。

PROGRAMS表示要产生的是可执行文件，有多个可执行文件时，用空格分开，而bin表示可执行文件的安装目录。SOURCES表示生成可执行文件需要的源文件，有多个源文件时，也用空格分开。

.am扩展名是automake的简称，它是automake用来产生Makefile.in文件的模板。

### 创建autoconf的模板

在helloworld下运行autoscan，生成文件configure.scan，把它改名为configure.in。这是autoconf的模板文件，它的内容大概为：

```
#                                               -*- Autoconf -*-
# Process this file with autoconf to produce a configure script.

AC_PREREQ(2.61)
AC_INIT(FULL-PACKAGE-NAME, VERSION, BUG-REPORT-ADDRESS)
AC_CONFIG_SRCDIR([src/main.c])
AC_CONFIG_HEADER([config.h])

# Checks for programs.
AC_PROG_CC

# Checks for libraries.

# Checks for header files.

# Checks for typedefs, structures, and compiler characteristics.

# Checks for library functions.

AC_CONFIG_FILES([Makefile
                 src/Makefile])
AC_OUTPUT
```

这个文件由一系列的宏组成，这些宏最终由命令m4展开，得到一个脚本文件configure。configure的主要功能是探测系统的配置，然后根据这些配置来产生相应的Makefile文件。比如AC_PROG_CC是用来检测编译器的，AC_CONFIG_FILES和AC_OUTPUT是用来产生Makefile和其他数据文件的。

不过这个模板文件还不能直接使用，需要做一点修改。把

```
AC_INIT(FULL-PACKAGE-NAME, VERSION, BUG-REPORT-ADDRESS)
```

改为

```
AC_INIT(helloworld, 0.1, xianjimli@hotmail.com)
```

FULL-PACKAGE-NAME是模块的名称。

VERSION是模块的版本号，初始版本号都用0.1。对小模块来说用两级版本号就够了，小数点前的为主版本号，只有重大更新时才升级主版本号。小数点后的为次版本号，每次发布都应该升级它。一般升级到0.9后，可以继续升级到0.10、0.11等。

BUG-REPORT-ADDRESS是作者或维护者的邮件地址。

还要再加上一行automake的初始化脚本。

```
AM_INIT_AUTOMAKE(helloworld, 0.1)
```

helloworld是模块的名称。

0.1是模块的版本号。

这里和前面的参数是重复的，AC_INIT是初始化autoconf的，AM_INIT_AUTOMAKE是初始automake的。在有的情况下，只是产生数据文件，而不需要编译文件时，那就不需要AM_INIT_AUTOMAKE了。

最后将得到下面的文件。

```
#                                                -*- Autoconf -*-
# Process this file with autoconf to produce a configure script.
AC_PREREQ(2.61)
AC_INIT(helloworld, 0.1, xianjimli@hotmail.com)
AC_CONFIG_SRCDIR([src/main.c])
AC_CONFIG_HEADER([config.h])
AM_INIT_AUTOMAKE(helloworld, 0.1)
# Checks for programs.
AC_PROG_CC
# Checks for libraries.
# Checks for header files.
# Checks for typedefs, structures, and compiler characteristics.
# Checks for library functions.
AC_CONFIG_FILES([Makefile
                 src/Makefile])
AC_OUTPUT
```

**复制所用到的宏**

运行：aclocal。

前面说了，configure.in里是一系列的宏，这些宏由命令m4负责展开。m4实际上是macro的简称，4代表m后面省略了4个字母。类似的还有I18n（Internationalization）和L10n（Localization），其中的数字都是代表所省略的字母个数。

AC_PROG_CC之类的宏是标准的宏（或说是内置的宏），不需要我们自己去写它，但我们需要运行命令aclocal，aclocal把configure.in中所用到的宏全部复制到我们的工程里来。在helloworld目录下运行aclocal之后，当前目录下出现了以下内容。

autom4te.cache——这是一个临时目录，只是用来加快宏展开的。

aclocal.m4——这是configure.in中用到的宏的定义，有兴趣的读者可以看看。

**生成配置头文件的模板**

运行：autoheader。

配置头文件（config.h）是用来定义在C/C++程序中可以引用的宏，像模块的名称和版本号等。这些宏由configure脚本产生，但我们要提供一个模板文件。这个模板文件可以用命令autoheader产生出来。在helloworld目录下运行autoheader之后，当前目录下产生config.h.in，一般情况不用修改它。

**创建几个必要的文件**

README：描述模块的功能、用法和注意事项等。

NEWS：描述模块最新的动态。

AUTHORS：模块的作者及联系方式。

ChangeLog：记录模块的修改历史，它有固定的格式：

(1) 最新修改放在最上面。

(2) 对于每条记录，第一行写日期，修改者和联系方式。第二行开始以tab开头（缩进），再加一个星号，后面再写修改的原因和位置等。如

```
2009-03-29 Li XianJing
    * Created
```

**生成Makefile.in和所需要的脚本**

运行：automake -a。

这个命令会建立COPYING depcomp INSTALL install-sh missing几个文件的链接，这些文件指向系统中的文件。automake最重要的功能是以Makefile.am为模板产生Makefile.in文件，Makefile.in相对于Makefile.am要复杂很多倍了，所幸的是我们不需要了解它。

**生成configure脚本**

运行：autoconf。

autoconf的功能是调用m4展开configure.in中的宏，生成configure脚本，这个脚本是最终运行的脚本。

**生成最终的Makefile**

运行：./configure。

configure有以下两个常用的参数。
--prefix 用来指定安装目录，Linux下默认的安装目录是/usr/local。
--host 用于交叉编译，比如x86的PC机上编译在ARM板上运行的程序。

如：

```
./configure --prefix=/home/lixianjing/work/arm-root/usr --host=arm-linux。
```

**编译**

运行：make。

**安装**

运行：make install。

**发布软件包**

运行：make dist或者make distcheck。

make dist用来生成一个发布软件包，这里会产生一个名为helloworld-0.1.tar.gz的文件。通常，源代码管理系统（cvs/svn/git）中的源代码是处于开发中的，是不稳定的，而发布的软件包则是稳定的，可供用户使用的。

怎样，是不是被这么多内容弄得有点晕了？其实我在这里主要是希望读者了解其中的原理，在实际操作中，我们可以把make之前的部分动作放到一个脚本文件中，只需要运行这个脚本就行了。这个脚本文件通常取名为autogen.sh或者bootstrap。

## 7.2 函数库

现在我们用automake来管理我们前面所建立的函数库，这是一个基础的函数库，我们就把它命名为base吧。

**目录结构**

base——根目录。
base/src ——源代码目录。

### 创建Makefile模板

base/Makefile.am内容如下。

```
SUBDIRS=src
```

base/src/Makefile.am内容如下。

```
lib_LTLIBRARIES=libbase.la

libbase_la_SOURCES= darray.c \
        darray.h \
        dlist.c \
        dlist.h \
        darray_iterator.h \
        dlist_iterator.h \
        hash_table.c \
        hash_table.h \
        invert.c \
        iterator.h \
        linear_container_darray.c \
        linear_container_darray.h \
        linear_container_dlist.c \
        linear_container_dlist.h \
        linear_container.h \
        queue.c \
        queue.h \
        sort.c \
        sort.h \
        stack.c \
        stack.h \
        typedef.h

libbase_la_LDFLAGS=-lpthread

noinst_PROGRAMS=darray_test dlist_test

darray_test_SOURCES=darray.c

darray_test_CFLAGS=-DDARRAY_TEST

dlist_test_SOURCES=dlist.c

dlist_test_CFLAGS=-DDLIST_TEST

basedir=$(includedir)/base

base_HEADERS=darray.h dlist.h iterator.h linear_container_dlist.h typedef.h \

        darray_iterator.h  dlist_iterator.h  linear_container_darray.h  \
```

```
        linear_container.h

EXTRA_DIST=\
        linear_container_test.c \
        invert_ng.c \
        darray_iterator.c \
        dlist_iterator.c \
        test_helper.c
```

LTLIBRARIES是关键字。LT代表libtool，libtool用来封装共享库在不同平台上的脚本，其具体实现我们不用关心。

libbase.la 是函数库的名称，扩展名用.la而不是.so或.a，它可以同时会生成共享库和静态库。libbase_la_SOURCES是生成libbase.la所需要的源文件。

LDFLAGS是用来指定链接时所需参数的关键字，-lpthread表示要链接libpthread.so。

noinst_PROGRAMS是关键字，表示不需要安装的可执行文件（通常是测试程序）。为了简单起见，这里没有写出全部的测试程序。

CFLAGS是关键字，用来指定编译和预处理时的参数。

HEADERS是关键字，列出所要安装的头文件。xxx_HEADERS和xxxdir要配套使用，后者表示要安装的位置。这里在base_HEADERS中列出的头文件会安装到basedir目录里。

**创建autoconf的模板**

运行

```
autoscan
mv configure.scan configure.in
```

然后按前面介绍的方法修改configure.in，得到下面的内容。

```
AC_PREREQ(2.61)
AC_INIT(base, 0.1, xianjimli@hotmail.com)
AC_CONFIG_SRCDIR([src/invert.c])
AC_CONFIG_HEADER([config.h])
AM_INIT_AUTOMAKE(base, 0.1)
# Checks for programs.
AC_PROG_CC
AC_PROG_LIBTOOL

# Checks for libraries.
# FIXME: Replace `main' with a function in `-lpthread':
AC_CHECK_LIB([pthread], [main])

# Checks for header files.
AC_HEADER_STDC
AC_CHECK_HEADERS([stdlib.h string.h unistd.h])
```

```
# Checks for typedefs, structures, and compiler characteristics.
AC_C_INLINE
AC_TYPE_SIZE_T

# Checks for library functions.
AC_FUNC_MALLOC
AC_FUNC_REALLOC

AC_CONFIG_FILES([Makefile
                 src/Makefile])
AC_OUTPUT
```

与前面不同的是：

(1) AC_PROG_LIBTOOL 用来检查libtool脚本；
(2) AC_CHECK_LIB用来检查共享库是否存在；
(3) AC_CHECK_HEADERS 用来检查头文件是否存在；
(4) AC_FUNC_MALLOC 用来检查标准的malloc函数是否存在。

**复制用到的宏。**

运行：aclocal。

**生成配置头文件的模板。**

运行：autoheader。

**创建README、NEWS、ChangeLog和AUTHORS几个文件。**

README：描述模块的功能、用法和注意事项等。
NEWS：描述模块最新的动态。
ChangeLog：记录模块的修改历史，它有固定的格式。
AUTHORS：模块的作者及联系方式。

**生成libtool需要的文件。**

运行：libtoolize --force --copy。

这个命令的主要功能是生成ltmain.sh，而ltmain.sh是用来产生libtool脚本的。

**生成Makefile.in和需要的脚本。**

运行：automake -a。

**生成configure脚本。**

运行：autoconf。

**生成最终的Makefile**

运行：./configure --prefix=$HOME/usr。

**编译**

运行：make。

**安装**

运行：make install。

**发布软件包**

运行：make dist。

我们编译好的文件安装到/home/lixianjing/usr/lib/目录下了。

```
libbase.a libbase.la libbase.so libbase.so.0 libbase.so.0.0.0
```

静态库：libbase.a。
动态库：libbase.so。

libtool的包装：libbase.la。
头文件和库都安装好了，调用者还需要知道下列信息才能使用。

头文件和库安装在哪里？
还依赖哪些其他模块？

为了解决这些问题，我们需要借助另外一个名为pkg-config的工具。pkg是package的简写，pkg-config负责查询指定软件包的配置信息，如软件包的名称、说明、版本号、头文件、库和依赖关系等等。为了让pkg-config能正常工作，软件包的实现者需要提供一个扩展名为pc 的配置文件。

pkg-config配置文件通常放在系统中的/usr/lib/pkgconfig/目录和/usr/local/lib/pkgconfig/目录下。下面是gtk+-2.0.pc。

```
prefix=/usr
exec_prefix=/usr
libdir=/usr/lib
```

```
includedir=/usr/include
target=x11

gtk_binary_version=2.10.0
gtk_host=i386-redhat-linux-gnu

Name: GTK+
Description: GIMP Tool Kit (${target} target)
Version: 2.12.10
Requires: gdk-${target}-2.0 atk cairo
Libs: -L${libdir} -lgtk-${target}-2.0
Cflags: -I${includedir}/gtk-2.0
```

前面部分是定义的一些变量，后面是一些关键字。

Name:名称。

Description:功能描述。

Version:版本号。

Requires:所依赖的软件包。

Libs:调用者的链接参数。

Cflags:调用者的编译参数。

由于prefix之类的变量是在软件包configure时才决定的，不能直接写死在pc文件中。我们可以让configure根据模板文件来产生。

模板文件名为base.pc.in，内容如下。

```
prefix=@prefix@
exec_prefix=${prefix}
libdir=${prefix}/lib
includedir=${prefix}/include

Name: @PACKAGE_NAME@
Description: a basic library.
Version: @VERSION@
Requires:
Libs: -L${libdir} -lbase
Cflags: -I${includedir}/base
```

这个模板文件和Makefile.in的替换规则一样，用两个@符号括起来的变量会替换成configure检测出来的值，@prefix@等变量是标准的变量。

修改一下base/Makefile.am，增加如下两行代码。

```
pkgconfigdir=${libdir}/pkgconfig
pkgconfig_DATA=base.pc
```

这是安装数据文件的方法，pkgconfig不是关键字，取个描述性的名称就好了。dir和_DATA是关键字，它们有相同的前缀，前者表示安装的目录，后者表示要安装的文件。按照惯例，pc文件将安装到${libdir}/pkgconfig目录下。

修改configure.in，增加输出文件base.pc。

```
AC_OUTPUT([base.pc])
```

放到AC_CONFIG_FILES也可以，它告诉configure脚本要产生的文件。

重新运行configure后会生成base.pc，内容如下。

```
prefix=/home/lixianjing/usr/local

exec_prefix=${prefix}

libdir=${prefix}/lib

includedir=${prefix}/include

Name: base

Description: a basic library.

Version: 0.1

Requires:

Libs: -L${libdir} -lbase

Cflags: -I${includedir}/base
```

(prefix与configure时指定的prefix参数一致。)

在下一节中，我们再学习调用者如何使用pc文件。

## 7.3 应用程序

前面我们创建的helloworld是一个很简单的应用程序工程，它只使用了标准C的函数。现在我们要建立一个应用程序工程，它将使用前面所写的libbase函数库。

**目录结构**

最顶层目录名用模块名称，这里用appdemo。

源文件放在模块下的src子目录里，即appdemo/src。

### 创建源文件

在src下创建源文件main.c，内容只是简单的调用一下libbase里的函数。

```
#include <dlist.h>

int main(int argc, char* argv[])
{
        DList* dlist = dlist_create(NULL, NULL);

        dlist_destroy(dlist);

        return 0;
}
```

### 创建Makefile模板

创建helloworld/Makefile.am，内容如下。

```
SUBDIRS=src
```

这里只有简单的一行代码，表示其下有一个src子目录，如果有多个子目录，用空格分开就行了。

创建helloworld/src/Makefile.am，内容如下。

```
bin_PROGRAMS=appdemo
appdemo_SOURCES=main.c
appdemo_CFLAGS=@BASE_CFLAGS@
appdemo_LDFLAGS=@BASE_LIBS@
```

appdemo_CFLAGS指定了编译appdemo时需要的参数，@BASE_CFLAGS@将替换成实际configure时所得到的参数。

appdemo_LDFLAGS是链接appdemo时需要的参数。@BASE_LIBS@将替换成实际configure时所得到的参数。

至于@BASE_CFLAGS@和@BASE_LIBS@是怎么得到的，后面我们再讲。

### 创建autoconf的模板

在appdemo下运行autoscan，生成文件configure.scan，把它改名为configure.in。这是autoconf的模板文件，它的内容大概如下。

```
#                                               -*- Autoconf -*-
# Process this file with autoconf to produce a configure script.
```

```
AC_PREREQ(2.61)
AC_INIT(FULL-PACKAGE-NAME, VERSION, BUG-REPORT-ADDRESS)
AC_CONFIG_SRCDIR([src/main.c])
AC_CONFIG_HEADER([config.h])

# Checks for programs.
AC_PROG_CC

# Checks for libraries.

# Checks for header files.

# Checks for typedefs, structures, and compiler characteristics.

# Checks for library functions.

AC_CONFIG_FILES([Makefile
                 src/Makefile])
AC_OUTPUT
```

按照前面介绍的方法，把configure.in的内容修改为：

```
#                                               -*- Autoconf -*-
# Process this file with autoconf to produce a configure script.

AC_PREREQ(2.61)
AC_INIT(appdemo, 0.1, xianjimli@hotmail.com)
AC_CONFIG_SRCDIR([src/main.c])
AC_CONFIG_HEADER([config.h])
AM_INIT_AUTOMAKE(appdemo, 0.1)

# Checks for programs.
AC_PROG_CC

# Checks for libraries.

# Checks for header files.

# Checks for typedefs, structures, and compiler characteristics.

# Checks for library functions.

AC_CONFIG_FILES([Makefile
                 src/Makefile])
AC_OUTPUT
```

与前面不同的是，我们还需要在# Checks for libraries. 之后添加检查libbase参数的宏。

```
PKG_CHECK_MODULES(BASE, ["base"])
AC_SUBST(BASE_CFLAGS)
AC_SUBST(BASE_LIBS)
```

PKG_CHECK_MODULES(BASE, ["base"]) 的功能是检查软件包base，通过调用前面所讲的pkg-config，生成 BASE_CFLAGS和 BASE_LIBS两个变量。

AC_SUBST(BASE_CFLAGS) 的功能是把所有对 BASE_CFLAGS的引用替换成实际的参数，即把Makefile.am中的@BASE_CFLAGS@替换成实际的值。

**复制所用到的宏**

运行：aclocal。

**生成配置头文件的模板**

运行：autoheader。

**创建几个必要的文件**

README：描述模块的功能、用法和注意事项等。
NEWS：描述模块最新的动态。
AUTHORS：模块的作者及联系方式。
ChangeLog：记录模块的修改历史，它有固定的格式。

**生成Makefile.in和所需要的脚本**

运行：automake -a。

**生成configure脚本**

运行：autoconf。

**生成最终的Makefile**

运行：./configure --prefix=$HOME/usr。

但这时会出现如下错误。

```
No package 'base' found

Consider adjusting the PKG_CONFIG_PATH environment variable if you
installed software in a non-standard prefix.
```

在默认情况下，pkg-config只是在/usr/lib/pkgconfig下查找相关的pc文件。如果pc文件安装在其他目录，需要设置环境变量 PKG_CONFIG_PATH。

```
export PKG_CONFIG_PATH=$HOME/usr/lib/pkgconfig
```

重新configure，一切都正常了。

**编译**

运行：make。

**安装**

运行：make install。

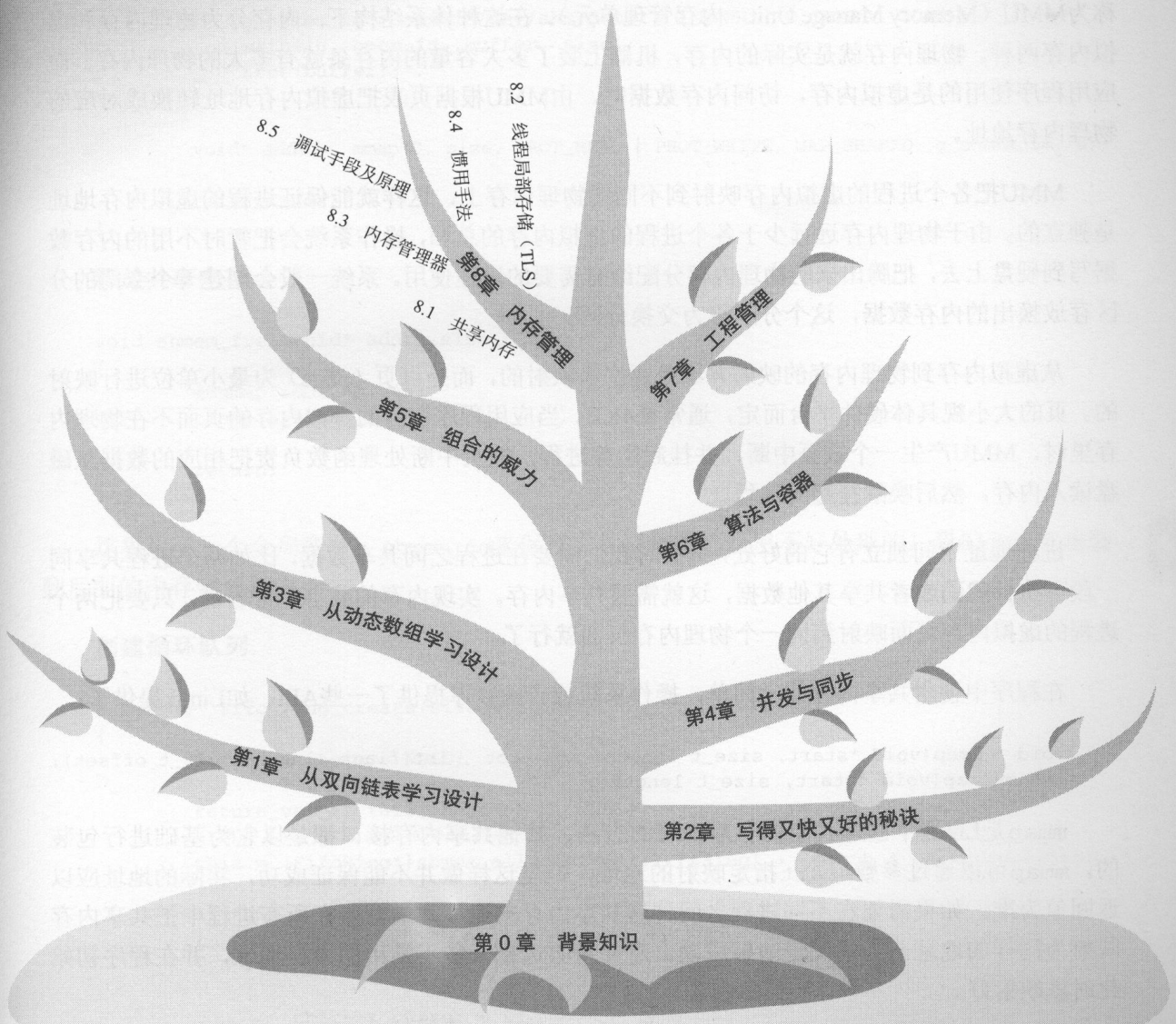
**发布软件包**

运行：make dist或make distcheck。

本章所讲的内容中可应付90%的情况了，如果需要使用更高级的特性，可以阅读相应的手册。

# 第 8 章

# 内存管理


8.2 线程局部存储（TLS）
8.4 惯用手法
8.5 调试手段及原理
8.3 内存管理器
第8章 内存管理
8.1 共享内存
第5章 组合的威力
第7章 工程管理
第6章 算法与容器
第3章 从动态数组学习设计
第4章 并发与同步
第1章 从双向链表学习设计
第2章 写得又快又好的秘诀
第 0 章 背景知识

## 8.1 共享内存

大家都知道，进程的地址空间是独立的，它们之间互不影响。比如同样地址为0xabcd1234的内存，在不同的进程中，它们的数据是完全不同的。这样做的好处有以下两点。

(1) 每个进程的地址空间变大了，编写程序更容易。

(2) 一个进程崩溃了，不会影响其他进程，提高了系统整体的稳定性。

要做到进程的地址空间独立，光靠软件是难以实现的，通常还依赖于硬件的帮助。这种硬件称为MMU（Memory Manage Unit，内存管理单元）。在这种体系结构下，内存分为物理内存和虚拟内存两种。物理内存就是实际的内存，机器上装了多大容量的内存条就有多大的物理内存。而应用程序使用的是虚拟内存，访问内存数据时，由MMU根据页表把虚拟内存地址转换成对应的物理内存地址。

MMU把各个进程的虚拟内存映射到不同的物理内存上，这样就能保证进程的虚拟内存地址是独立的。由于物理内存远远少于各个进程的虚拟内存的总和，操作系统会把暂时不用的内存数据写到硬盘上去，把腾出来的物理内存分配给有需要的进程使用。系统一般会创建一个专门的分区存放换出的内存数据，这个分区称为交换分区。

从虚拟内存到物理内存的映射并不是逐字节映射的，而是以页（page）为最小单位进行映射的。页的大小视具体硬件平台而定，通常是4KB。当应用程序访问的虚拟内存的页面不在物理内存里时，MMU产生一个缺页中断，并挂起当前进程，缺页中断处理函数负责把相应的数据从磁盘读入内存，然后唤醒挂起的进程。

进程地址空间独立有它的好处，但有时我们需要在进程之间共享数据，比如两个进程共享同一段可执行代码或者共享其他数据，这就需要共享内存。实现内存的共享非常容易，只要把两个进程的虚拟内存页面映射到同一个物理内存页面就行了。

在程序中使用共享内存非常简单，操作系统或者函数库提供了一些API。如Linux提供了：

```
void * mmap(void *start, size_t length, int prot , int flags, int fd, off_t offset);
int munmap(void *start, size_t length);
```

mmap是Linux下创建共享内存最正统的方法，其他共享内存接口都是以它为基础进行包装的。mmap可以通过参数start指定映射的地址，但是这样做并不能保证成功，实际的地址应以返回值为准。如果需要在不同进程之间传递共享内存的地址，最好是在所有进程中把共享内存映射为同样的地址。为了保证映射成功，需要精心选择一些不常用的内存地址，并在程序初始化时就映射好。

接下来将讲解一个示例，我们用下循环队列（FIFO ring）在两个进程之间传递数据，由于循环队列在一个读一个写的情况下不需要加锁，我们暂时可以避开进程间同步的问题。

### 创建共享内存

```
static int g_shmem_fd = -1;
void* shmem_alloc(size_t size)
{
        g_shmem_fd = open("/dev/shm/demo", O_RDWR);

        if(g_shmem_fd < 0)
        {
            char* buffer = calloc(size, 1);
            g_shmem_fd = open("/dev/shm/demo", O_RDWR | O_CREAT, 0640);
            write(g_shmem_fd, buffer, size);
            free(buffer);
        }

        void* addr =  mmap(0, size, PROT_READ | PROT_WRITE, MAP_SHARED, g_shmem_fd, 0);

        return addr;
}
```

### 释放共享内存

```
void shmem_free(void* addr, size_t size)
{
        munmap(addr, size);
        close(g_shmem_fd);

        return;
}
```

这里用了一个全局变量g_shmem_fd来保存文件描述符。这里不好处理的，我们会把它封装到后面的内存管理器中。

### 创建循环队列

```
FifoRing* fifo_ring_create(size_t length)
{
        FifoRing* thiz = NULL;

        return_val_if_fail(length > 1, NULL);
        /*用共享内存分配函数代替malloc*/
        thiz = (FifoRing*)shmem_alloc(sizeof(FifoRing) + length * sizeof(void*));

        if(thiz != NULL)
        {
            if(thiz->inited == 0)
            {
                thiz->r_cursor = 0;
```

```
            thiz->w_cursor = 0;
            thiz->length   = length;
            thiz->inited = 1;
        }
    }

    return thiz;
}
```

这里的shmem_alloc代替了以前的malloc。

### 销毁循环队列

```
void fifo_ring_destroy(FifoRing* thiz)
{
    if(thiz != NULL)
    {
        shmem_free(thiz, sizeof(FifoRing) + thiz->length * sizeof(void*));
    }

    return;
}
```

这里的shmem_free代替了以前的free。

循环队列的其他函数实现不变。

### 取数据

```
Ret fifo_ring_pop(FifoRing* thiz, void** data)
{
    Ret ret = RET_FAIL;
    return_val_if_fail(thiz != NULL && data != NULL, RET_FAIL);
    if(thiz->r_cursor != thiz->w_cursor)
    {
        *data = thiz->data[thiz->r_cursor];
        thiz->r_cursor = (thiz->r_cursor + 1)%thiz->length;
        ret = RET_OK;
    }
    return ret;
}
```

### 追加数据

```
Ret fifo_ring_push(FifoRing* thiz, void* data)
{
    int w_cursor = 0;
    Ret ret = RET_FAIL;
    return_val_if_fail(thiz != NULL, RET_FAIL);
    w_cursor = (thiz->w_cursor + 1) % thiz->length;
    if(w_cursor != thiz->r_cursor)
    {
```

```
            thiz->data[thiz->w_cursor] = data;
            thiz->w_cursor = w_cursor;
            ret = RET_OK;
        }
        return ret;
}
```

## 8.2 线程局部存储（TLS）

同一个进程中的多个线程，它们的内存空间是共享的（栈除外），一个线程对内存的修改，对所有线程都有效。这既是一个优点也是一个缺点。说它是优点，因为这样会让线程间的数据交换快捷高效。说它是缺点，一个线程死掉了，其他线程也将性命不保。

在Unix下，大家一直都对线程不太感兴趣，直到很晚才引入真正的线程。像X Sever要同时处理*N*个客户端的连接，每秒钟要响应上百万个请求，开发人员宁愿自己实现调度机制也不用线程。再如Apache（1.3x版），在Unix下的实现也是采用多进程的。

正如《Unix编程艺术》[①]中所说，线程局部存储的出现，使得这种情况出现了转机。采用线程局部存储，每个线程有一定的私有空间。这可以避免部分无意的破坏（当然无法避免有意的破坏行为），也省去线程访问共享数据时出现的问题。

其实这完全是因为Unix程序不喜欢面向对象方法引起的，数据没有很好地封装起来，全局变量满天飞，在多线程情况下自然容易出问题。如果采用面向对象的方法，情况将大为改观，甚至不需要线程局部存储来帮忙。

不过，多一种技术就多一种选择，知道线程局部存储肯定没有坏处。在有的情况下，没有线程局部存储，确实很难用其他办法实现。比如，glibc会把最后一次操作的错误码记录在errno变量里，如果不采用线程局部存储，在多线程的情况下，当前线程取的errno可能不是自己上一次操作的错误码。

线程局部存储在不同的平台上有不同的实现，可移植性不太好。幸好要实现线程局部存储并不难，最简单的办法就是建立一个全局表，通过当前线程ID去查询相应的数据，因为各个线程的ID不同，查到的数据自然也不同了。

大多数平台都提供了线程局部存储的方法，无需要我们自己去实现，在Linux下有两种方法可以实现线程局部存储。

**方法一：使用pthread的函数**

```
int pthread_key_create(pthread_key_t *key, void (*destructor)(void*));
```

① Eric Raymond著。该书已由电子工业出版社于2006年出版。

```
int pthread_key_delete(pthread_key_t key);
void *pthread_getspecific(pthread_key_t key);
int pthread_setspecific(pthread_key_t key, const void *value);
```

示例如下。

```
static pthread_key_t key;
static pthread_once_t key_once = PTHREAD_ONCE_INIT;

static void make_key(void)
{
    pthread_key_create(&key, NULL);

    return;
}

void* thread_entry(void* param)
{
    pthread_once(&key_once, make_key);

    if (pthread_getspecific(key) == NULL)
    {
        pthread_setspecific(key, (void*)pthread_self());
    }

    printf("data=%u\n", pthread_getspecific(key));

    return NULL;
}
```

pthread_once保证make_key只被执行一次。

**方法二：使用编译器扩展**

示例如下：

```
__thread int i;
```

方法二使用起来方便得多，但前者的好处是移植比较方便，即使编译器本身不支持，也可以自己封装相应的函数。

## 8.3 内存管理器

在前面学习共享内存的时候，我们重新实现了循环队列，两个实现的不同之处只是在于内存分配和释放上。对比一下 fifo_ring_create的实现。

**第一种实现：用malloc分配内存。**

```
FifoRing* fifo_ring_create(size_t length)
```

```
{
        FifoRing* thiz = NULL;

        return_val_if_fail(length > 1, NULL);

        thiz = (FifoRing*)malloc(sizeof(FifoRing) + length * sizeof(void*));

        if(thiz != NULL)
        {
                thiz->r_cursor = 0;
                thiz->w_cursor = 0;
                thiz->length   = length;
        }

        return thiz;
}
```

**第二种实现：用shmem_alloc分配内存。**

```
FifoRing* fifo_ring_create(size_t length)
{
    FifoRing* thiz = NULL;

    return_val_if_fail(length > 1, NULL);

    thiz = (FifoRing*)shmem_alloc(sizeof(FifoRing) + length * sizeof(void*));

    if(thiz != NULL)
    {
                    if(thiz->inited == 0)
                    {
            thiz->r_cursor = 0;
            thiz->w_cursor = 0;
            thiz->length   = length;
            thiz->inited = 1;
                }
    }

    return thiz;
}
```

只是一行代码之差，就要把整个队列重写一遍，不符合我们前面倡导的DRY原则。这里可以看出，内存管理器有不同的实现，所以我们引入内存管理器这个接口来隔离变化。内存管理器的基本功能有：

- 分配内存；
- 释放内存；
- 扩展/缩小已经分配的内存；
- 分配清零的内存。

据此，我们定义Allocator接口如下。

```
struct  _Allocator;
typedef struct  _Allocator Allocator;

typedef void* (*AllocatorCallocFunc)(Allocator* thiz, size_t nmemb, size_t size);
typedef void* (*AllocatorAllocFunc)(Allocator* thiz, size_t size);
typedef void  (*AllocatorFreeFunc)(Allocator* thiz, void *ptr);
typedef void* (*AllocatorReallocFunc)(Allocator* thiz, void *ptr, size_t size);
typedef void  (*AllocatorDestroyFunc)(Allocator* thiz);

struct _Allocator
{
    AllocatorCallocFunc  calloc;
    AllocatorAllocFunc   alloc;
    AllocatorFreeFunc    free;
    AllocatorReallocFunc realloc;
    AllocatorDestroyFunc destroy;

    char priv[0];
};
```

基于malloc系统函数的实现。

```
static void*  allocator_normal_calloc(Allocator* thiz, size_t nmemb, size_t size)
{
  return calloc(nmemb, size);
}
  ...
Allocator* allocator_normal_create(void)
{
  Allocator* thiz = (Allocator*)calloc(1, sizeof(Allocator));

  if(thiz != NULL)
  {
              thiz->calloc  = allocator_normal_calloc;
              thiz->alloc   = allocator_normal_alloc;
              thiz->realloc = allocator_normal_realloc;
              thiz->free    = allocator_normal_free;
              thiz->destroy = allocator_normal_destroy;
  }

  return thiz;
}
```

这里的函数与标准C函数一一对应，只需要简单包装就行了。

对于共享内存，通常的做法是先分配一大块内存，然后进行二次分配，此时需要编写自己的**内存管理器**。内存分配器是很奇特的，任何初学者都可以设计自己的内存分配器，但同时任何高手都不敢说自己能设计出最好的内存分配器。为什么内存分配器很难写好呢？因为设计好的内存分配器需要考虑很多因素。

❑ 最大化的兼容性。

实现内存分配器时，先要定义出分配器的接口函数。接口函数没有必要标新立异，而是要遵循现有标准（如POSIX或者Win32），让使用者可以平滑地过渡到新的内存分配器上。

❑ 最大化的可移植性。

通常情况下，内存分配器要向操作系统申请内存，然后进行二次分配，这要调用操作系统提供的函数才行。操作系统提供的函数则是因平台而异，尽量抽象出平台相关的代码，才能保证内存分配器的可移植性。

❑ 浪费的空间最小化。

内存分配器要管理内存，必然要使用一些自己的数据结构，这些数据结构本身也要占内存空间。在用户眼中，这些内存空间毫无疑问是浪费掉了，如果浪费在内存分配器本身的内存太多，显然是不可以接受的。而内存碎片也是浪费空间的罪魁祸首，内存碎片是一些不连续的小块内存，它们总量可能很大，但因为其非连续性而无法使用，这些空间在某种程度上说也是浪费的。

❑ 最快的速度。

内存分配/释放是常用的操作。按着2/8原则，常用函数的性能对系统的整体性能影响最大，所以内存分配器的速度越快越好。遗憾的是，最先匹配算法、最优匹配算法和最差匹配算法，谁也不能说谁更快，因为快与慢要依赖于具体的应用环境。

❑ 最大化的可调性（以适应于不同的情况）。

内存管理算法设计的难点就在于要适应不同的情况。事实上，如果缺乏应用的上下文，是无法评估内存管理算法的好坏的，因为专用算法总是在时/空性能上有更优的表现。为每种情况都写一套内存管理算法，显然是不太合适的。我们不需要追求最优算法，那样代价太高，能达到次优就行了。设计一套通用内存管理算法，通过一些参数对它进行配置，可以让它在特定情况也有相当出色的表现，这就是可调性。

❑ 最大化的调试功能。

作为一个C/C++程序员，内存错误可以说是我们的噩梦，上一次的内存错误一定还让你记忆犹新。内存分配器提供的调试功能，强大易用，特别是对嵌入式环境来说，在内存错误检测工具缺乏的情况下，内存分配器提供的调试功能就更不可少了。

❑ 最大化的适应性。

前面说了最大化可调性，以便让内存分配器适用于不同的情况。但是，对于不同情况都要去调整配置，还是有点麻烦。尽量让内存分配器适用于很广的情况，只在极少情况下才去调整配置，这就是内存分配器的适应性。

设计是一个多目标优化的过程，而且有些目标之间存在着竞争。如何平衡这些竞争是设计的

难点之一。在不同的情况下，这些目标的重要性是不一样的，所以根本不存在一个最好的内存分配器。换句话说，内存分配器的实现是变化的，要根据不同的应用环境而变化。

下面我们实现一个傻瓜型的内存管理器，按上面的标准来看，它没有什么实际的意义，但它的优点是简单，仅仅200多行代码，就展示了内存管理器的基本原理。

### 分配过程

所有空闲的内存块放在一个双向链表中，最初只有一块。分配时使用首次匹配算法，在第一个空闲块上进行分配。

```
static void*  allocator_self_manage_alloc(Allocator* thiz, size_t size)
{
    FreeNode* iter = NULL;
    size_t length = REAL_SIZE(size);
    PrivInfo* priv = (PrivInfo*)thiz->priv;

    /*查找第一个满足条件的空闲块*/
    for(iter = priv->free_list; iter != NULL; iter = iter->next)
    {
                        if(iter->length > length)
                        {
                break;
                        }
    }

    return_val_if_fail(iter != NULL, NULL);

    /*如果找到的空闲块刚好满足需求，就从空闲块链表中移出它*/
    if(iter->length < (length + MIN_SIZE))
    {
                        if(priv->free_list == iter)
                        {
                priv->free_list = iter->next;
                        }

                        if(iter->prev != NULL)
                        {
                iter->prev->next = iter->next;
                        }
                        if(iter->next != NULL)
                        {
            iter->next->prev = iter->prev;
                        }
    }
    else
    {
                        /*如果找到的空闲块比较大，就把它拆成两个块，把多余的空闲内存放回去*/
                        FreeNode* new_iter = (FreeNode*)((char*)iter + length);
```

```
                    new_iter->length = iter->length - length;
                    new_iter->next = iter->next;
                    new_iter->prev = iter->prev;

                    if(iter->prev != NULL)
                    {
            iter->prev->next = new_iter;
                    }
                    if(iter->next != NULL)
                    {
            iter->next->prev = new_iter;
                    }

                    if(priv->free_list == iter)
                    {
            priv->free_list = new_iter;
                    }
                            iter->length = length;
    }
    /*返回可用的内存*/
    return (char*)iter + sizeof(size_t);
}
```

这里对空闲块的管理不占用有效内存空间，它只是把空闲块强制转换成FreeNode结构，这要求任何空闲块的大小都不小于sizeof(FreeNode)。

### 释放内存

释放时把内存块放回空闲链表，然后对相邻居的内存块进行合并。

```
static void   allocator_self_manage_free(Allocator* thiz, void *ptr)
{
    FreeNode* iter = NULL;
    FreeNode* free_iter = NULL;
    PrivInfo* priv = (PrivInfo*)thiz->priv;

    return_if_fail(ptr != NULL);

    free_iter = (FreeNode*)((char*)ptr - sizeof(size_t));

    free_iter->prev = NULL;
    free_iter->next = NULL;

    if(priv->free_list == NULL)
    {
                    priv->free_list = free_iter;

                    return;
    }
    /*根据内存块地址的大小，把它插入到适当的位置。*/
    for(iter = priv->free_list; iter != NULL; iter = iter->next)
    {
                    if((size_t)iter > (size_t)free_iter)
```

```
                {
            free_iter->next = iter;
            free_iter->prev = iter->prev;
            if(iter->prev != NULL)
            {
                iter->prev->next = free_iter;
            }
            iter->prev = free_iter;

            if(priv->free_list == iter)
            {
                priv->free_list = free_iter;
            }

            break;
                }

                if(iter->next == NULL)
                {
            iter->next = free_iter;
            free_iter->prev = iter;

            break;
            }
    }

    /*对相邻居的内存进行合并*/
    allocator_self_manage_merge(thiz, free_iter);

    return;
}

/*合并相邻的两个块*/
static void allocator_self_manage_merge2(Allocator* thiz, FreeNode* prev, FreeNode* next)
{
    PrivInfo* priv = (PrivInfo*)thiz->priv;
    return_if_fail(prev != NULL && next != NULL && prev->next == next);

    prev->next = next->next;
    if(next->next != NULL)
    {
                next->next->prev = prev;
    }
    prev->length += next->length;

    if(priv->free_list == next)
    {
                priv->free_list = prev;
    }

    return;
}

/*递归地合并相邻的两个块*/
```

```
static void allocator_self_manage_merge(Allocator* thiz, FreeNode* iter)
{
    FreeNode* prev = iter->prev;
    FreeNode* next = iter->next;

    /*与前面的块合并*/
    if(prev != NULL && ((size_t)prev + prev->length) == (size_t)iter)
    {
                        allocator_self_manage_merge2(thiz, prev, iter);
                        allocator_self_manage_merge(thiz, prev);
    }
                        /*与后面的块合并*/
    if(next != NULL && ((size_t)iter + iter->length) == (size_t)next)
    {
        allocator_self_manage_merge2(thiz, iter, next);
        allocator_self_manage_merge(thiz, iter);
    }

    return;
}
```

有了Allocator接口，我们也可以通过装饰模式，为内存管理器加上越界/泄露检查等其他辅助功能。下面的Allocator是检查内存越界的装饰。

**实现函数**

```
static void*  allocator_checkbo_alloc(Allocator* thiz, size_t size)
{
    char* ptr = NULL;
    size_t total = size + 3 * sizeof(void*);
    PrivInfo* priv = (PrivInfo*)thiz->priv;

    if((ptr = allocator_alloc(priv->real_allocator, total)) != NULL)
    {
        /*记录长度，前后的MAGIC*/
                    *(size_t*)ptr = size;
                    memset(ptr + sizeof(void*), BEGIN_MAGIC, sizeof(void*));
                    memset(ptr + 2 * sizeof(void*) + size, END_MAGIC, sizeof(void*));
    }

    return ptr + 2 * sizeof(void*);
}
```

分配内存时：多分配3 * sizeof(void*)个字节，分别用来记录内存块的长度及前后的Magic数据。然后调用实际的（即被装饰的）内存分配器分配内存，记录长度并填充Magic数据，最后返回内存指针给调用者。

```
static void   allocator_checkbo_free(Allocator* thiz, void *ptr)
{
    PrivInfo* priv = (PrivInfo*)thiz->priv;
```

```
    if(ptr != NULL)
    {
        /*释放时检查MAGIC是否有变化*/
            char magic[sizeof(void*)];
            char* real_ptr = (char*)ptr - 2 * sizeof(void*);
            size_t size = *(size_t*)(real_ptr);

            memset(magic, BEGIN_MAGIC, sizeof(void*));
            assert(memcmp((char*)ptr - sizeof(void*), magic, sizeof(void*)) == 0);
            memset(magic, END_MAGIC, sizeof(void*));
            assert(memcmp((char*)ptr + size , magic, sizeof(void*)) == 0);

            allocator_free(priv->real_allocator, real_ptr);
    }

    return;
}
```

释放内存时：先取得内存块的长度，然后检查前后的magic数据是否被修改，如果被修改则认为存在写越界的错误。

## 8.4 惯用手法

*Pattern-Oriented Software Architecture*[①]（面向模式的软件架构）一书中根据模式粒度把模式分为三类：架构模式、设计模式和惯用手法。其中把分层模式、管道过滤器和微内核模式等归为架构模式，把代理模式、命令模式和出版-订阅模式等归为设计模式，而把引用计数等归为惯用手法。这三类模式间的界限比较模糊，在特定的情况下，有的设计模式可以作为架构模式来用，有的架构模式也能作为设计模式来用。

一般来说，我们可以认为架构模式、设计模式和惯用手法，三者的重要性依次递减，毕竟整体决策比局部决策的影响面更大。但是任何整体都是由局部组成的，局部的决策也会影响整体。惯用手法的影响虽然通常是局部的，但其所起的作用仍然很重要。本文介绍几种关于内存使用的惯用手法。

**预分配**

在前面的学习中，我们已经用到了这种手法。在实现一个动态数组时，如果向其中增加元素，它便会自动扩展（缩减）缓冲区的大小，而不需要调用者关心。扩展缓冲区的过程如下。

(1) 先分配一块更大的缓冲区。
(2) 把数据从老的缓冲区复制到新的缓冲区。
(3) 释放老的缓冲区。

① 原书由Wiley出版。——编者注

如果你使用realloc来实现，内存管理器可能会做些优化：如果老的缓冲区后面有连续的空闲空间，它只需要扩展老的缓冲区，而跳过后面两个步骤。但在大多数情况下，它都要通过上述三个步骤来完成扩展。

由此可见，扩展缓冲区对调用者来说虽然是透明的，但决不是免费的。它得付出相当大的时间代价，以及由此产生的产生内存碎片问题。如果每次向vector中增加一个元素都要扩展缓冲区，显然是不太合适的。

此时我们可以采用预分配机制，每次扩展时，不是需要多大就扩展多大，而是预先分配一大块内存。这一大块可以供后面较长一段时间使用，直到把这块内存全用完了，再继续用同样的方式扩展。

至于一次应该扩展多大，经验值是现有大小的1.5倍。这个经验值基于这样的假设：现在用得多意味着将来会用得更多。

预分配机制多见于一些带缓冲区的容器实现中，比如vector和string等。

**对象引用计数**

在面向对象的系统中，对象之间的协作关系非常复杂。所谓协作其实就调用对象的函数或者向对象发送消息，但不管调用函数还是发送消息，总是要通过某种方式知道目标对象才行。而最常见的做法就是保存目标对象的引用（指针）。

对象被别人引用了，但自己可能并不知道。此时麻烦就来了，如果对象被销毁了，对该对象的引用就变成了野针，系统随时可能因此而崩溃。不释放也不行，因为那样会出现内存泄露。怎么办呢？

此时我们可以采用对象引用计数，对象有一个引用计数器，不管谁要引用这个对象，就要把对象的引用计数器加1，如果不再引用了，就把对象的引用计数器减1。当对象的引用计数器被减为0时，说明没有其他对象引用它，该对象就可以安全地销毁了。这样，对象的生命周期就得到了有效的管理。

还有一种引用称为弱引用，它不增加对象的引用计数，只是要求对象在销毁时通知引用者。实现方法是调用者注册一个回调函数，在对象销毁时调用。

对象引用计数运用相当广泛。像在Windows下COM（组件对象模型）和GNOME的glib里，对象引用计数都是对象系统的基本设施之一。

**写时复制**

操作系统内核创建子进程的过程是最常见而且最有效的写时复制（COW，Copy on Write）

的例子：创建子进程时，子进程要继承父进程内存空间中的数据。但继承之后，两者各自有独立的内存空间，修改各自的数据不会互相影响。

要做到这一点，最简单的办法就是直接把父进程的内存空间复制一份。这样做可行，但问题在于要复制的内容太多，无论是时间还是空间上的开销都让人无法接受。况且，在大多数情况下，子进程只会使用少数继承过来的数据，而且多数是读取，只有少量是修改，也就说大部分复制的动作是无用功。怎么办呢？

此时可以采用写时复制。最初的复制只是个假象，并不是真正进行复制，只是把引用计数加1，并设置适当的标志。如果双方都只是读取这些数据，那好办，直接读就行了。而任何一方要修改时，为了不影响另外一方，它要把数据复制一份，然后修改复制的这一份数据。也就是说在修改数据时，复制动作才真正发生。

当然，在真正复制的时候，你可以选择是复制一部分数据还是复制全部数据。在上面的例子中，由于内存通常是按页来管理的，因此复制时只复制相关的页，而不是复制整个内存空间。

写时复制对性能的贡献很大，差不多任何带MMU的操作系统都会采用。当然它不限于内核空间，在用户空间也可以使用，比如像一些`String`类的实现也采用了这种方法。

**固定大小分配**

频繁地分配大量小块内存是设计内存管理器的挑战之一。

首先是空间利用率上的问题：由于内存管理器本身需要一些辅助内存，假设每块内存需要16字节用作辅助内存，那么即使只要分配4个字节这样的小块内存，仍然要分配16字节的内存。一块小内存不要紧，若存在大量小块内存，所浪费的空间就不容忽视了。

其次是内存碎片问题：频繁分配大量小块内存，很容易造成内存碎片问题。这不但降低内存管理器的效率，同时由于这些内存不连续，即便处于空闲也无法使用。

此时可以采用固定大小分配，这种方式通常也叫做缓冲池（pool）分配。缓冲池先分配一块或者多块连续的大块内存，把它们分成*N*块大小相等的小块内存，然后进行二次分配。由于这些小块内存的大小是固定的，内存管理的开销就非常小了，往往只要一个标识位用于标识该单元是否空闲，或者甚至连标识位都不需要。

缓冲池中所有这些小块内存分布在一块或者几块连续内存上，所以不会有内存碎片问题。

固定大小分配手法运用比较广泛，差不多所有的内存管理器都用这种方法来对付小块内存分配，比如glibc、STLPort和Linux的slab等。

**会话缓冲池分配**

服务器要长时间运行，内存泄露是它的威胁之一，任何小概率的内存泄露，都可能会累积到具有破坏性的程度。从它们的运行模式来看，它们总是不断地重复某个过程，而在这个过程中，又要大量（多次）分配内存。

比如Web服务器，它不断地处理HTTP请求。我们把一次HTTP请求称为一次会话，一次会话要经过多个阶段，在这个过程中要做各种处理，要多次分配内存。由于处理比较复杂，分配内存的地方又比较多，内存泄露可以说是防不甚防。

针对这种情况，我们可以采用会话缓冲池分配。它基于多次分配一次释放的策略，在过程开始时创建会话缓冲池（session pool），这个过程中所有内存分配都通过会话缓冲池来分配，当这个过程完成时，销毁掉会话缓冲池，即释放这个过程中所分配的全部内存。

因为只需要释放一次，内存泄露的可能大大降低了。会话缓冲池分配并不是太常见，Apache采用的就是这种用法。我自己用过几次，感觉效果不错。

还有一些关于内存使用的惯用手法，这里不再多说了，有兴趣的读者可以参考相关资料。

## 8.5 调试手段及原理

这里我们从应用程序、编译器和调试器三个层次来防止和排查内存错误。了解这些知识，一方面可以满足我们的求知欲，另一方面也确实能给我们带来不少帮助。在不同的层次，有不同的方法，这些方法有各自的长处和局限，要学会在不同的场合使用不同的方法。

**从应用程序的层次**

最好的情况是从设计到编码都扎扎实实，避免把错误引入到程序中来，这才是解决问题的根本之道。但问题在于，理想情况并不存在，现实中存在大量具有内存错误的程序，如果内存错误很容易避免，Java和C#的优势将不会那么突出了。虽然我们不指望在编码时，能够避免所有内存错误，但这些努力是最有价值的。

对于内存错误，应用程序自己能做的非常有限。但由于这类内存错误非常典型，占的比例很大，我们付出的努力与所得的回报相比而言还是非常划算的，因此仍然值得我们去研究。

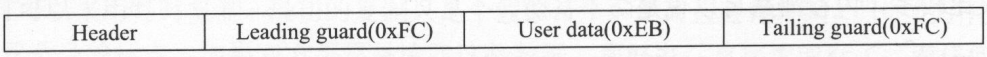
前面我们讲过，堆里面的内存是由内存管理器管理的。从应用程序的角度来看，我们能做到的就是打内存管理器的主意。其实原理很简单。

对付内存泄露。重载内存管理函数，在分配时，把这块内存的记录到一个链表中，在释放时，从链表中删除掉，在程序退出时，检查链表是否为空，如果不为空，则说明有内存泄露，否则说

明没有泄露。当然，为了查出是哪里的泄露，在链表还要记录是谁分配的，通常记录文件名和行号就行了。

对付内存越界/野指针。对这两者，我们只能检查一些典型的情况，对其他一些情况无能为力，但效果仍然不错。其方法如下（参考自*Comparing and contrasting the runtime error detection technologies*）。

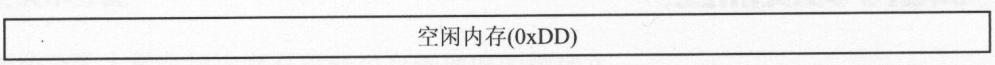
(1) 首尾再加保护边界值。




在内存分配时，内存管理器按如上结构填充分配出来的内存。其中Header是管理器自己用的，前后各有几个字节的guard数据，它们的值是固定的。当内存释放时，内存管理器检查这些guard数据是否被修改，如果被修改，说明有写越界。

它的工作机制注定了有它的局限性：只能检查写越界，不能检查读越界，而且只能检查连续性的写越界，对于跳跃性的写越界无能为力。不过由于连续的写越界是常见的，所以这种方法仍有其价值。

(2) 填充空闲内存。




内存被释放之后，它的内容填充成固定的值。这样，从指针指向的内存的数据，可以大致判断这个指针是否是野指针。

它同样有它的局限：程序要主动判断才行。如果野指针指向的内存立即被重新分配了，它又被填充成前面那个结构，这时也无法检查出来。

**从编译器的层次**

Boundschecker和Purify是两个著名的内存错误检查工具，它们的实现都可以归于编译器一级。前者采用一种称为CTI（Compile-Time Instrumentation）的技术。编译器的编译不是要分几个阶段吗？Boundschecker在预处理和编译两个阶段之间，对源文件进行修改。它对所有内存分配释放、内存读写、指针赋值和指针计算等所有内存相关的操作进行分析，并插入自己的代码。比如：

修改前

```
if (m_hsession) gblHandles->ReleaseUserHandle( m_hsession );
  if (m_dberr) delete m_dberr;
```

修改后

```
if (m_hsession) {
    _Insight_stack_call(0);
    gblHandles->ReleaseUserHandle(m_hsession);
    _Insight_after_call();
}

_Insight_ptra_check(1994, (void **) &m_dberr, (void *) m_dberr);

if (m_dberr) {
    _Insight_deletea(1994, (void **) &m_dberr, (void *) m_dberr, 0);
    delete m_dberr;
}
```

Purify则采用一种称为OCI（Object Code Insertion）的技术。与Boundschecker的CTI技术不同的是，它会对可执行文件的每条指令进行分析，找出所有内存分配释放、内存读写、指针赋值和指针计算等所有与内存相关的操作，并用自己的指令代替原始的指令。

Boundschecker和Purify是商业软件，它们的实现是保密的、拥有专利的，我们无法对其进行研究，只能找一些皮毛性的介绍。无论是CTI还是OCI这样的名称，对我们而言多少都有些神秘。其实它们的实现原理并不复杂，通过对Valgrind和gcc的bounds checker扩展进行一些粗浅的研究，我们可以了解它们的大致原理。

gcc的bounds checker基本上可以与Boundschecker对应起来，都是对源代码进行修改，以达到控制内存操作功能，如`malloc/free`等内存管理函数、`memcpy/strcpy/memset`等内存读取函数和指针运算等。Valgrind则与Purify类似，都是通过对目标代码进行修改，来达到同样的目的。

Valgrind对可执行文件进行修改，所以不需要重新编译程序。但它并不是在执行前对可执行文件和所有相关的共享库进行一次性修改，而是和应用程序在同一个进程中运行，动态地修改即将执行的下一段代码。

Valgrind是插件式设计的。Core部分负责对应用程序的整体控制，并把即将修改的代码，转换成一种中间格式，这种格式类似于RISC指令，然后把中间代码传给插件。插件根据要求对中间代码修改，然后把修改后的结果交给Core。Core接下来把修改后的中间代码转换成原始的x86指令，并执行它。

由此可见，无论是Boundschecker、Purify、gcc的bounds checker，还是Valgrind，修改源代码也罢，修改二进制也罢，都是对代码进行修改。究竟要修改什么，修改成什么样子呢？别急，下面我们就来介绍。

- 管理所有内存块。无论是堆、栈还是全局变量，只要有指针引用它，它就被记录到一个全局表中。记录的信息包括内存块的起始地址和大小等。管理所有内存块其实并不难：

对于在堆里分配的动态内存，可以通过重载内存管理函数来实现。对于全局变量等静态内存，可以从符号表中得到这些信息。

- 拦截所有的指针计算。对指针进行乘除等运算通常意义不大，最常见运算是对指针加减一个偏移量，如++p、p=p+n和p=a[n]等。所有这些有意义的指针操作都要受到检查，检查操作不再是由一条简单的汇编指令来完成，而是由一个函数来完成。

有了以上两点保证，要检查内存错误就非常容易了。比如要检查++p是否有效，首先在全局表中查找p指向的内存块，如果没有找到，说明p是野指针。如果找到了，再检查p+1是否在这块内存范围内，如果不是，那就是越界访问，否则就是正常的。怎么样？够简单吧？无论是全局内存、堆还是栈，无论是读还是写，没有内存错误能够逃过工具的法眼。

**代码赏析（源于tcc）**

对指针运算进行检查

```
void *__bound_ptr_add(void *p, int offset)
{
        unsigned long addr = (unsigned long)p;

        BoundEntry *e;

    #if defined(BOUND_DEBUG)

        printf("add: 0x%x %d\n", (int)p, offset);

    #endif

        e = __bound_t1[addr >> (BOUND_T2_BITS + BOUND_T3_BITS)];
        e = (BoundEntry *)((char *)e +
                           ((addr >> (BOUND_T3_BITS - BOUND_E_BITS)) &
                        ((BOUND_T2_SIZE - 1) << BOUND_E_BITS)));

        addr -= e->start;
        if (addr > e->size) {
            e = __bound_find_region(e, p);

            addr = (unsigned long)p - e->start;
        }

        addr += offset;

        if (addr > e->size)
            return INVALID_POINTER; /* return an invalid pointer */

        return p + offset;
}

static void __bound_check(const void *p, size_t size)
```

```
{
        if (size == 0)
            return;

        p = __bound_ptr_add((void *)p, size);

        if (p == INVALID_POINTER)
        bound_error("invalid pointer");
}
```

重载内存管理函数

```
void *__bound_malloc(size_t size, const void *caller)
{
        void *ptr;

        /* we allocate one more byte to ensure the regions will be
        separated by at least one byte. With the glibc malloc, it may
        be in fact not necessary */

        ptr = libc_malloc(size + 1);

        if (!ptr)

            return NULL;

        __bound_new_region(ptr, size);

        return ptr;
}

void __bound_free(void *ptr, const void *caller)
{
        if (ptr == NULL)
            return;

        if (__bound_delete_region(ptr) != 0)
        bound_error("freeing invalid region");

        libc_free(ptr);
}
```

重载内存操作函数

```
void *__bound_memcpy(void *dst, const void *src, size_t size)
{
        __bound_check(dst, size);

        __bound_check(src, size);

        /* check also region overlap */
```

```
        if (src >= dst && src < dst + size)
            bound_error("overlapping regions in memcpy()");
        return memcpy(dst, src, size);
}
```

**从调试器的层次**

现在有OS的支持，实现一个调试器变得非常简单，至少原理不再神秘。这里我们简要介绍一下Win32和Linux中的调试器实现原理。

在Win32下，实现调试器主要通过两个函数：WaitForDebugEvent和ContinueDebugEvent。下面是一个调试器的基本模型（参考自*Debugging Applications for Microsoft .NET and Microsoft Windows*[①]）。

```
void main ( void )
{
        CreateProcess ( ..., DEBUG_ONLY_THIS_PROCESS ,... ) ;
    while ( 1 == WaitForDebugEvent ( ... ) )
        {
            if ( EXIT_PROCESS )
            {
                break ;
            }
            ContinueDebugEvent ( ... ) ;
        }
}
```

调试器会启动被调试的进程，并指定DEBUG_ONLY_THIS_PROCESS标志。按Win32下事件驱动的一贯原则，被调试的进程应该主动上报调试事件，然后再由调试器做相应的处理。

在Linux下，实现调试器只要ptrace函数就行了。下面是个简单示例（参考自*Playing with ptrace*）。

```
#include <sys/ptrace.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>
#include <linux/user.h>   /* For user_regs_struct
                             etc. */

int main(int argc, char *argv[])
{
    pid_t traced_process;
```

① 原书由Microsoft Press于2003年出版。——编者注

```
        struct user_regs_struct regs;

        long ins;

        if(argc != 2) {
            printf("Usage: %s <pid to be traced>\n",
                   argv[0], argv[1]);

            exit(1);
        }

        traced_process = atoi(argv[1]);

        ptrace(PTRACE_ATTACH, traced_process,
            NULL, NULL);

        wait(NULL);

        ptrace(PTRACE_GETREGS, traced_process,
             NULL, &regs);

        ins = ptrace(PTRACE_PEEKTEXT, traced_process,
                     regs.eip, NULL);

        printf("EIP: %lx Instruction executed: %lx\n",
               regs.eip, ins);

        ptrace(PTRACE_DETACH, traced_process,
               NULL, NULL);

        return 0;
}
```

最后在介绍几个调试器常见操作的实现方法。

- 设置断点：设置断点很简单，先通过源代码位置（如main函数）找到二进制代码的位置（如0x80483e2），然后在这个位置写一条中断指令（在x86上为0xcc）即可，CPU执行到这条指令时就会发生中断。我们可以用gdb做个小实验。

```
int test(int a, int b)
{
        return a+b;
}

int main(int argc, char* argv[])
{
        int i = 0;
        unsigned int* p = (unsigned int*)test;

        printf("content of function test: %08x\n", *p);
```

```
        return 0;
}
```

正常情况下这个程序的输出如下。

```
content of function test: 8be58955
```

在gdb中运行，在函数test处设置一个断点，则会输出以下内容。

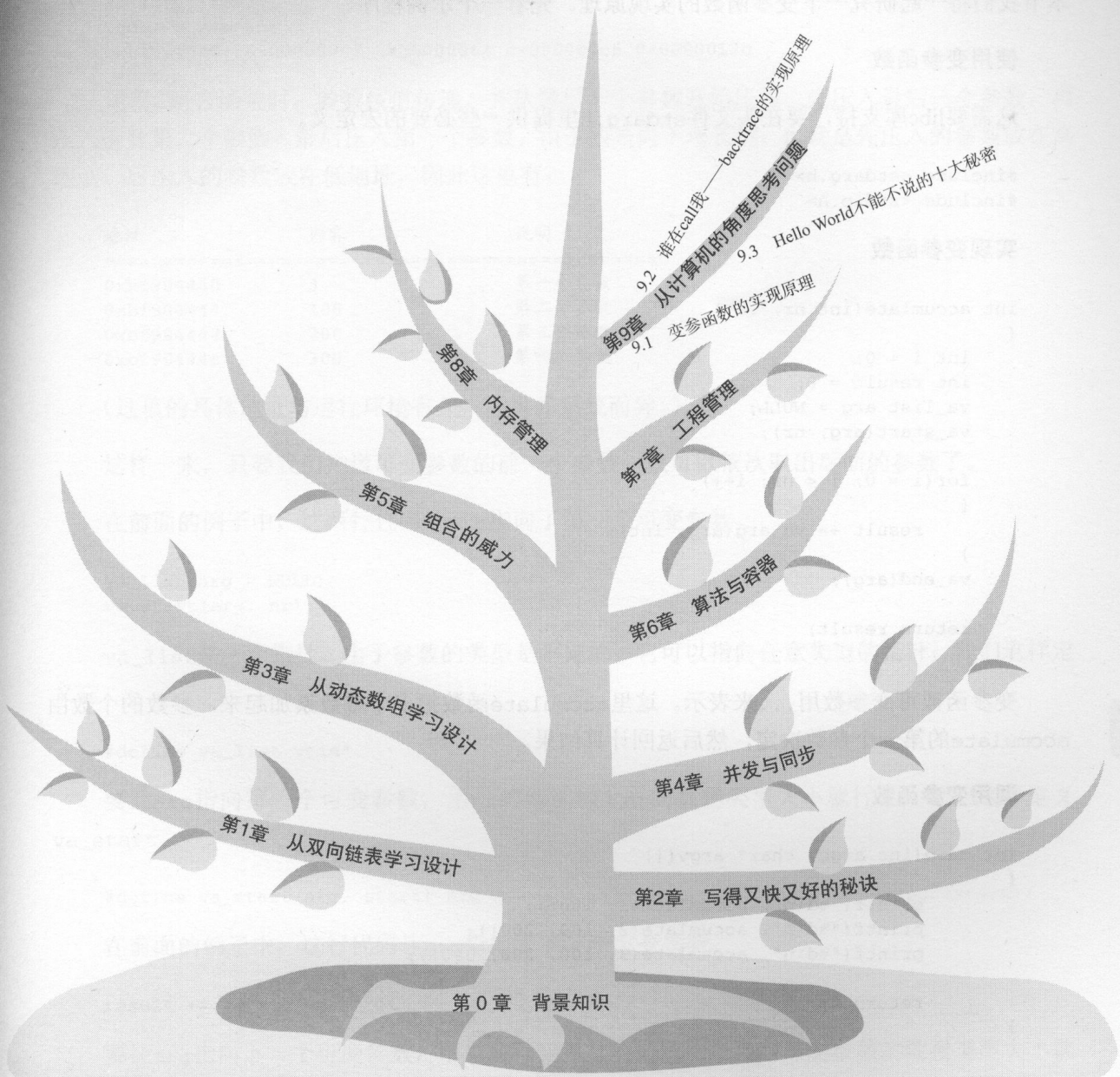
```
content of function test: cce58955
```

这说明设置断点之后，8b被修改为cc。

- 显示变量的内容：先查找符号表，得到变量名对应的内存地址，然后获取内存中的数据，最后按变量的类型显示出来。
- 显示寄存器的内容：ptrace的PTRACE_GETREGS调用可以得到寄存器的内容。

# 第 9 章

# 从计算机的角度思考问题



## 9.1 变参函数的实现原理

C语言要求函数调用者按照函数原型进行调用，如果调用参数与函数原型不一致，编译器就会发出警告。而变参函数的参数是不确定的，它允许同一个函数有多种不同的参数组合，编译器不会对可变部分的参数做类型检查，因而在使用的时候拥有较大的灵活性（当然也容易出错）。本节我们将一起研究一下变参函数的实现原理，先看一个示例程序。

### 使用变参函数

这需要libc库支持，要在头文件stdarg.h里提供一些必要的宏定义。

```
#include <stdarg.h>
#include <stdio.h>
```

### 实现变参函数

```
int accumlate(int nr, ...)
{
    int i = 0;
    int result = 0;
    va_list arg = NULL;
    va_start(arg, nr);

    for(i = 0; i < nr; i++)
    {
        result += va_arg(arg, int);
    }
    va_end(arg);

    return result;
}
```

变参函数可变参数用...来表示。这里accmulate函数把多个整数累加起来，参数的个数由accmulate的第一个参数决定，然后返回计算结果。

### 调用变参函数

```
int main(int argc, char* argv[])
{
        printf("%d\n", accumlate(1, 100));
        printf("%d\n", accumlate(2, 100, 200));
        printf("%d\n", accumlate(3, 100, 200, 300));

        return 0;
}
```

这里以三种方式调用了accumlate。

在上面的例子中，va_list/ va_start/ va_arg/ va_end几个宏完成了所有的实现细节。那它们到底是怎么实现的呢？我们先看看参数在内存里的布局。

用gdb调试上述程序，在函数accmulate里设置断点，然后显示nr相邻区域的数据。

```
Breakpoint 1, accumlate (nr=3) at varg.c:13
13 int i = 0;
(gdb) x /4w &nr
0xbf904440: 0x00000003 0x00000064 0x000000c8 0x0000012c
```

调用C语言函数时，参数按值传递，并从最后一个参数开始压栈。先压入最后一个参数，再压入倒数第二个参数，最后压入第一个参数。由于栈是向下增长的，也就是先压入的参数放在高地址，后压入的参数放在低地址，因此这里有：

| 地址 | 内容 | 说明 |
| --- | --- | --- |
| 0xbf904440 | 3 | 第一个参数 |
| 0xbf904444 | 100 | 第二个参数 |
| 0xbf904448 | 200 | 第三个参数 |
| 0xbf90444c | 300 | 第四个参数 |

（这里的具体地址与运行环境有关，因操作系统而异。）

这样一来，只要我们知道可变参数的前一个参数，就可以依次取出后面的参数了。

在前面的例子中，这两行代码让arg指向了第一个可变参数。

```
va_list arg = NULL;
va_start(arg, nr);
```

va_list是一个指针，由于参数的类型是不定的，它可以指向任意类型的指针，我们这样定义它。

```
#define va_list void*
```

要让arg指向第一个可变参数，用nr的地址加上nr的数据类型大小就行了，我们这样定义va_start。

```
#define va_start(arg, start) arg = (va_list)(((char*)&(start)) + sizeof(start))
```

在前面的例子中，这行代码让arg指向了下一个可变参数。

```
result += va_arg(arg, int)
```

要让arg指向下一个可变参数，用当前可变参数的地址加上当前可变参数的数据类型大小就

行了，我们这样定义 va_arg。

```
#define va_arg(arg, type)    *(type*)arg; arg = (char*)arg + sizeof(type);
```

可变参数的实现原理简单吧，其实即使没有标准C的支持，我们也可以实现变参函数。

```
#include <stdio.h>

#define va_list void*
#define va_end(arg)
#define va_arg(arg, type)    *(type*)arg; arg = (char*)arg + sizeof(type);
#define va_start(arg, start) arg = (va_list)(((char*)&(start)) + sizeof(start))

int accumlate(int nr, ...)
{
        int i = 0;
        int result = 0;
        va_list arg = NULL;
        va_start(arg, nr);

        for(i = 0; i < nr; i++)
        {
            result += va_arg(arg, int);
        }
        va_end(arg);

        return result;
}

int main(int argc, char* argv[])
{
        printf("%d\n", accumlate(1, 100));
        printf("%d\n", accumlate(2, 100, 200));
        printf("%d\n", accumlate(3, 100, 200, 300));

        return 0;
}
```

对于变参函数，我还要说明几点。

**编译器优化**

如果加了编译优化标志，参数可能是通过寄存器传递的，那参数在内存里的布局与前面所展示的就不一样了。这个倒不用担心，因为编译器会处理的，它会将变参函数的所有参数保存到内存里。我们可以看一下ARM平台上的汇编代码。

```
00000000 <accmulate>:
   0:    e92d000f     stmdb    sp!, {r0, r1, r2, r3}
   4:    e59dc000     ldr    ip, [sp]
   8:    e35c0000     cmp    ip, #0    ; 0x0
   c:    d3a00000     movle  r0, #0    ; 0x0
```

```
10:    da000008    ble    38 <accmulate+0x38>
14:    e3a00000    mov    r0, #0    ; 0x0
18:    e28d1008    add    r1, sp, #8    ; 0x8
1c:    e1a02000    mov    r2, r0
20:    e5113004    ldr    r3, [r1, #-4]
24:    e2822001    add    r2, r2, #1    ; 0x1
28:    e15c0002    cmp    ip, r2
2c:    e0800003    add    r0, r0, r3
30:    e2811004    add    r1, r1, #4    ; 0x4
34:    1afffff9    bne    20 <accmulate+0x20>
38:    e28dd010    add    sp, sp, #16    ; 0x10
3c:    e12fff1e    bx     lr
```

r0、r1、r2和r3四个寄存器通常用来传递函数的前面四个参数，编译器在这里对此做了特殊处理，用stmdb sp!, {r0, r1, r2, r3}这条指令把r0到r3的四个寄存器的值保存到内存里了。

**关于参数结束标识的问题**

变参函数的参数个数是变化的，怎么知道实际参数的个数呢？通常的做法有三种。

(1) 指定参数的个数。比如这里accmulate的第一个参数指明了参数的个数。
(2) 用固定值（如-1或NULL）表示最后一个参数。
(3) 用格式化字符串。比如printf使用了格式化字符串。

**变参函数至少要提供一个参数**

这很明显，如果没有可变参数前面的一个参数，我们就无法取得第一个可变参数的地址。

## 9.2 谁在call我——backtrace的实现原理

显示函数调用关系（backtrace/callstack）是调试器必备的功能之一，比如在gdb里，用bt命令就可以查看backtrace。在程序崩溃的时候，函数调用关系有助于快速定位问题的根源，了解它的实现原理，可以扩充自己的知识面，在没有调试器的情况下，也能实现自己backtrace。更重要的是，分析backtrace的实现原理很有意思。现在我们一起来研究一下：

glibc提供了一个backtrace函数，这个函数可以帮助我们获取当前函数的backtrace，先看看它的使用方法，后面我们再仿照它写一个。

```
#include <stdio.h>
#include <stdlib.h>
#include <execinfo.h>

#define MAX_LEVEL 4
```

```
static void test2()
{
    int i = 0;
    void* buffer[MAX_LEVEL] = {0};

    int size = backtrace(buffer, MAX_LEVEL);

    for(i = 0; i < size; i++)
    {
        printf("called by %p\n",  buffer[i]);
    }

    return;
}

static void test1()
{
    int a=0x11111111;
    int b=0x11111112;

    test2();
    a = b;

    return;
}

static void test()
{
    int a=0x10000000;
    int b=0x10000002;

    test1();
    a = b;

    return;
}

int main(int argc, char* argv[])
{
    test();

    return 0;
}
```

用以下命令编译运行它。

```
gcc -g -Wall bt_std.c -o bt_std
./bt_std
```

屏幕打印出如下结果。

```
called by 0×8048440
called by 0×804848a
```

```
called by 0x80484ab
called by 0x80484c9
```

上面打印的是调用者的地址，对程序员来说也许不太直观，glib还提供了另外一个函数backtrace_symbols，它可以把这些地址转换成源代码的位置（通常是函数名）。不过这个函数并不怎么好用，特别是在没有调试信息的情况下，几乎得不到什么有用的信息。这里我们使用另外一个工具addr2line来实现地址到源代码位置的转换。

运行

```
./bt_std |awk '{print "addr2line "$3" -e bt_std"}'>t.sh;. t.sh;rm -f t.sh
```

屏幕打印

```
/home/work/mine/sysprog/think-in-compway/backtrace/bt_std.c:12
/home/work/mine/sysprog/think-in-compway/backtrace/bt_std.c:28
/home/work/mine/sysprog/think-in-compway/backtrace/bt_std.c:39
/home/work/mine/sysprog/think-in-compway/backtrace/bt_std.c:48
```

backtrace是如何实现的呢？在x86的机器上，在函数被调用时，栈中数据的结构如下。

```
-----------------------------------------------
参数 N
参数 …          函数参数入栈的顺序与具体的调用方式有关
参数 3
参数 2
参数 1
-----------------------------------------------
EIP             完成本次调用后，下一条指令的地址
EBP             保存调用者的EBP，然后EBP指向此时的栈顶
----------------新的EBP指向这里----------------
临时变量1
临时变量2
临时变量3
临时变量…
临时变量5
-----------------------------------------------
```

（说明：下面是低地址，上面是高地址，栈是向下增长的。）

在函数被调用时，先把被调函数的参数压入栈中，C语言的压栈方式是：先压入最后一个参数，再压入倒数第二参数，按此顺序入栈，最后才压入第一个参数。

然后压入EIP和EBP，此时EIP指向完成本次调用后下一条指令的地址，我们可以近似地认为这个地址是函数调用者的地址。EBP是调用者和被调函数之间的分界线，分界线之上是调用者的临时变量、被调函数的参数、函数返回地址（EIP），和上一层函数的EBP，分界线之下是被调函数的临时变量。

最后压入被调函数本身，并为它分配临时变量的空间。不同版本gcc的处理方式是不一样的，对于老版本的gcc（如gcc 3.4），第一个临时变量放在最高的地址，第二个其次，依次顺序分布。而对于新版本的gcc（如gcc 4.3），临时变量的位置是反的，即最后一个临时变量在最高的地址，倒数第二个其次，依次逆序分布。

为了实现backtrace，我们需要：

(1) 获取当前函数的EBP；
(2) 通过EBP获得调用者的EIP；
(3) 通过EBP获得上一级的EBP；
(4) 重复这个过程，直到结束。

通过嵌入汇编代码，我们可以获得当前函数的EBP，不过这里我们不用汇编，而且通过临时变量的地址来获得当前函数的EBP。我们知道，对于gcc3.4生成的代码，当前函数第一个临时变量的下一个位置就是EBP。而对于gcc4.3生成的代码，当前函数最后一个临时变量的下一个位置就是 EBP。

有了这些背景知识，我们来实现自己的backtrace。

```
#ifdef NEW_GCC
#define OFFSET 4
#else
#define OFFSET 0
#endif/*NEW_GCC*/

int backtrace(void** buffer, int size)
{
        int  n = 0xfefefefe;
        int* p = &n;
        int  i = 0;

        int ebp = p[1 + OFFSET];
        int eip = p[2 + OFFSET];

        for(i = 0; i < size; i++)
        {
            buffer[i] = (void*)eip;
            p = (int*)ebp;
            ebp = p[0];
            eip = p[1];
        }

        return size;
}
```

对于老版本的gcc，OFFSET定义为0，此时p+1就是EBP，而p[1]就是上一级的EBP，p[2]是

调用者的EIP。本函数总共有5个 int类型的临时变量，所以对于新版本gcc， OFFSET定义为5，此时p+5就是EBP。在一个循环中，重复取上一层的EBP和EIP，最终得到所有调用者的EIP，从而实现了 backtrace。

现在我们用完整的程序来测试一下bt.c。

```
#include <stdio.h>

#define MAX_LEVEL 4
#ifdef NEW_GCC
#define OFFSET 4
#else
#define OFFSET 0
#endif/*NEW_GCC*/

int backtrace(void** buffer, int size)
{
    int     n = 0xfefefefe;
    int* p = &n;
    int     i = 0;

    int ebp = p[1 + OFFSET];
    int eip = p[2 + OFFSET];

    for(i = 0; i < size; i++)
    {
        buffer[i] = (void*)eip;
        p = (int*)ebp;
        ebp = p[0];
        eip = p[1];
    }

    return size;
}

static void test2()
{
    int i = 0;
    void* buffer[MAX_LEVEL] = {0};

    backtrace(buffer, MAX_LEVEL);

    for(i = 0; i < MAX_LEVEL; i++)
    {
        printf("called by %p\n",    buffer[i]);
    }

    return;
}
static void test1()
{
```

```
    int a=0x11111111;
    int b=0x11111112;

    test2();
    a = b;

    return;
}

static void test()
{
    int a=0x10000000;
    int b=0x10000002;

    test1();
    a = b;

    return;
}

int main(int argc, char* argv[])
{
    test();

    return 0;
}
```

写个简单的Makefile。

```
CFLAGS=-g -Wall
all:
        gcc34 $(CFLAGS) bt.c -o bt34
        gcc $(CFLAGS) -DNEW_GCC  bt.c -o bt
        gcc $(CFLAGS) bt_std.c -o bt_std

clean:
        rm -f bt bt34 bt_std
```

编译然后运行以下命令。

```
make
./bt|awk '{print " addr2line " $3" -e bt" }'>t.sh;. t.sh;
```

屏幕打印如下结果。

```
/home/work/mine/sysprog/think-in-compway/backtrace/bt.c:37
/home/work/mine/sysprog/think-in-compway/backtrace/bt.c:51
/home/work/mine/sysprog/think-in-compway/backtrace/bt.c:62
/home/work/mine/sysprog/think-in-compway/backtrace/bt.c:71
```

对于可执行文件，这种方法能起作用。但对于共享库，addr2line就无法根据这个地址找到对应的源代码位置了。这是因为addr2line只能通过地址偏移量来查找，而打印出的地址是绝对

地址。由于共享库加载到内存的位置是不确定的，为了计算地址偏移量，我们还需要进程maps文件的帮助。

通过进程的maps文件（/proc/进程号/maps），我们可以找到共享库的加载位置。

```
...
00c5d000-00c5e000 r-xp 00000000 08:05 2129013
/home/work/mine/sysprog/think-in-compway/backtrace/libbt_so.so
00c5e000-00c5f000 rw-p 00000000 08:05 2129013
/home/work/mine/sysprog/think-in-compway/backtrace/libbt_so.so
...
```

libbt_so.so的代码段加载到0×00c5d000-0×00c5e000，而backtrace打印出的地址是：

```
called by 0xc5d4eb
called by 0xc5d535
called by 0xc5d556
called by 0×80484ca
```

这里可以用打印出的地址减去加载的地址来计算偏移量。如，用0xc5d4eb减去加载地址0×00c5d000，得到偏移量0×4eb，然后把0×4eb传给addr2line。

```
addr2line 0×4eb -f -s -e ./libbt_so.so
```

屏幕会打印如下结果。

```
/home/work/mine/sysprog/think-in-compway/backtrace/bt_so.c:38
```

栈里的数据很有意思，在上一节中，通过分析栈里的数据，我们了解了变参函数的实现原理。在这一节中，通过分析栈里的数据，我们又学到了backtrace的实现原理。

## 9.3 Hello World 不能不说的十大秘密

Hello World是最经典的入门程序，该程序因Brian Kernighan和Dennis Ritchie编写的《C程序设计语言》[①]（The C Programming Language）而广泛流传。Hello World同样也是深入研究计算机的极好题材，可以说我对计算机的理解，很大程度上归功于对Hello World的研究。后来看了《深入理解计算机》和台湾著名黑客黄敬群老师的《深入浅出Hello World》之后，对Hello World又有了更深的认识。这里和大家分享一下Hello World背后的秘密。

C语言的Hello World如下。

```
#include <stdio.h>
```

① 《C程序设计语言（第2版）》已由机械工业出版社出版。——编者注

```
int main(int argc, char* argv[])
{
        printf("Hello World!\n");

        return 0;
}
```

**秘密一：main函数的原型**

有些初学者是这样写Hello World的，编译也可以通过，运行也正常：

```
void main(void)
{
        printf("Hello World!\n");

        return;
}
```

如果用gcc来编译，你会发现，把main写成什么样子都行，只要函数名为main，编译都可以通过（可能有警告）。但下面几种写法才是比较正规的。

```
int main(void)
int main(int argc, char* argv[])
int main(int argc, char* argv[], char* env[])
```

argc是命令行参数的个数。

argv是命令行参数，以NULL结束。

env是环境变量，以NULL结束。

下面的程序可以显示argv和env的内容。

```
#include <stdio.h>

int main(int argc, char* argv[], char* env[])
{
    int i = 0;
    printf("Hello World!\n");

    for(i = 0; argv[i] != NULL; i++)
    {
        printf("argv[%d]=%s\n", i, argv[i]);
    }

    for(i = 0; env[i] != NULL; i++)
    {
        printf("env[%d]=%s\n", i, env[i]);
    }

    return 0;
}
```

编译后运行它：

```
./helloworld arg1 arg2
```

屏幕会打印：

```
Hello World!
argv[0]=./helloworld
argv[1]=arg1
argv[2]=arg2
env[0]=SSH_AGENT_PID=2609
env[1]=HOSTNAME=lixj.linux
env[2]=DESKTOP_STARTUP_ID=
env[3]=TERM=xterm
...
```

环境变量是从父进程继承过来的，通过setenv和getenv等函数可以存取环境变量，但对环境变量的修改只会影响当前进程及子进程，而不会影响父进程。

**秘密二：main函数的返回值**

正常情况下，我们调用一个函数之后，通过检查它的返回值来判断函数执行的结果。但main函数不是由程序员自己调用的，那么它的返回值会返回给谁呢？

答案是：main函数的返回值是返回给父进程的，父进程调用下列函数来获取子进程的退出码（即main函数的返回值）。

```
pid_t wait(int *status);
pid_t waitpid(pid_t pid, int *status, int options);
```

在bash里，执行一个命令后（bash是父进程，命令是子进程），$?里存放的是这个命令的退出码，我们来测试一下。

```
int main(int argc, char* argv[])
{
        printf("Hello World!\n");

        return 100;
}
```

编译运行：

```
./helloworld_2;echo $?
```

屏幕打印：

```
Hello World!
100
```

100正是我们的返回值。这里对上面的程序做点修改，让它返回1000，看看它真的会返回1000吗？

```
int main(int argc, char* argv[])
{
    printf("Hello World!\n");

    return 1000;
}
```

编译运行：

```
./helloworld_2;echo $?
```

屏幕打印：

```
Hello World!
232
```

奇怪的是，这里打印的不是1000，而是232！不少朋友都碰到过类似的问题，我自己也遇到过。后来查资料才知道，main函数的返回值虽然是int的，可以保存32位的整数，但实际上，系统只使用了一个字节来保存返回值，所以这是打印的是232（即1000 & 0xff）。

**秘密三：被隐藏的细节**

现在我们用strace来分析一下helloworld的执行过程。

```
strace ./helloworld_3
```

屏幕会打印：

```
execve("./helloworld_3", ["./helloworld_3"], [/* 51 vars */]) = 0
brk(0)                                  = 0x8eda000
mmap2(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0xb8029000
access("/etc/ld.so.preload", R_OK)      = -1 ENOENT (No such file or directory)
open("./tls/i686/sse2/libc.so.6", O_RDONLY) = -1 ENOENT (No such file or directory)
open("./tls/i686/libc.so.6", O_RDONLY)  = -1 ENOENT (No such file or directory)
open("./tls/sse2/libc.so.6", O_RDONLY)  = -1 ENOENT (No such file or directory)
open("./tls/libc.so.6", O_RDONLY)       = -1 ENOENT (No such file or directory)
open("./i686/sse2/libc.so.6", O_RDONLY) = -1 ENOENT (No such file or directory)
open("./i686/libc.so.6", O_RDONLY)      = -1 ENOENT (No such file or directory)
open("./sse2/libc.so.6", O_RDONLY)      = -1 ENOENT (No such file or directory)
open("./libc.so.6", O_RDONLY)           = -1 ENOENT (No such file or directory)
open("/etc/ld.so.cache", O_RDONLY)      = 3
fstat64(3, {st_mode=S_IFREG|0644, st_size=123031, ...}) = 0
mmap2(NULL, 123031, PROT_READ, MAP_PRIVATE, 3, 0) = 0xb800a000
close(3)                                = 0
open("/lib/libc.so.6", O_RDONLY)        = 3
read(3, "\177ELF\1\1\1\0\0\0\0\0\0\0\0\0\3\0\3\0\1\0\0\0@\307o\0004\0\0\0"..., 512) = 512
```

```
    fstat64(3, {st_mode=S_IFREG|0755, st_size=1758448, ...}) = 0
    mmap2(0x6e6000, 1476176, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_DENYWRITE, 3, 0) =
0x6e6000
    mmap2(0x849000, 12288, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3,
0x163) = 0x849000
    mmap2(0x84c000, 9808, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_ANONYMOUS, -1,
0) = 0x84c000
    close(3)                                = 0
    mmap2(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) =
0xb8009000
    set_thread_area({entry_number:-1 -> 6, base_addr:0xb80096c0, limit:1048575,
seg_32bit:1, contents:0, read_exec_only:0, limit_in_pages:1, seg_not_present:0,
useable:1}) = 0
    mprotect(0x849000, 8192, PROT_READ)     = 0
    mprotect(0x6e2000, 4096, PROT_READ)     = 0
    munmap(0xb800a000, 123031)              = 0
    fstat64(1, {st_mode=S_IFCHR|0620, st_rdev=makedev(136, 1), ...}) = 0
    mmap2(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) =
0xb8028000
    write(1, "Hello World!\n", 13Hello World!)          = 13
    exit_group(0)                           = ?
```

这么一行简单的程序，居然做数十次系统调用。可见简单的程序并不简单，只是实现细节被操作系统和函数库封装起来罢了。

前面只是加载并执行helloworld程序，真正打印字符串的是write函数：它把字符串写入文件描述符为1的文件里。在C语言中：

(1) 文件描述符0 表示标准输入；

(2) 文件描述符1 表示标准输出；

(3) 文件描述符2 表示标准错误输出。

**秘密四：printf不见了**

我们看下main函数的汇编代码（x86）。

```
int main(int argc, char* argv[], char* env[])
{
 80483b4:    8d 4c 24 04             lea    0x4(%esp),%ecx
 80483b8:    83 e4 f0                and    $0xfffffff0,%esp
 80483bb:    ff 71 fc                pushl  -0x4(%ecx)
 80483be:    55                      push   %ebp
 80483bf:    89 e5                   mov    %esp,%ebp
 80483c1:    51                      push   %ecx
 80483c2:    83 ec 04                sub    $0x4,%esp
    printf("Hello World!\n");
 80483c5:    c7 04 24 a4 84 04 08    movl   $0x80484a4,(%esp)
 80483cc:    e8 1f ff ff ff          call   80482f0 <puts@plt>
```

```
    return 0;
 80483d1:    b8 00 00 00 00              mov    $0x0,%eax
}
```

奇怪的是这里并没有调用printf，而且是调用的puts。第一次见到这个代码，我想printf可能只是个宏，最终由puts来实现打印功能。不过后来证实glibc里确实有printf函数，那为什么这里变成了puts呢？

我对代码做了修改，使它不包含任何头文件，并自己声明printf的函数原型，这样确保没有宏在作怪。

```
int printf(const char *format, ...);

int main(int argc, char* argv[], char* env[])
{
        printf("Hello World!\n");

        return 0;
}
```

反汇编出来的代码没有任何变化，由此可见，不是宏在做怪，而是gcc做了手脚。原因可能是：printf要对格式字符进行分析，相对来说效率低下，如果只有一个参数，printf的功能和puts一致，于是gcc就用puts代替了它。为了证实这个观点，对代码再做一点修改，使用格式字符串打印。

```
#include <stdio.h>

int main(int argc, char* argv[], char* env[])
{
        printf("%s%s", "Hello", " World!\n");

        return 0;
}
```

这次汇编代码变成：

```
int main(int argc, char* argv[], char* env[])
{
 80483c4:    8d 4c 24 04                 lea    0x4(%esp),%ecx
 80483c8:    83 e4 f0                    and    $0xfffffff0,%esp
 80483cb:    ff 71 fc                    pushl  -0x4(%ecx)
 80483ce:    55                          push   %ebp
 80483cf:    89 e5                       mov    %esp,%ebp
 80483d1:    51                          push   %ecx
 80483d2:    83 ec 14                    sub    $0x14,%esp
    printf("%s%s", "Hello", " World!\n");
 80483d5:    c7 44 24 08 c4 84 04        movl   $0x80484c4,0x8(%esp)
 80483dc:    08
 80483dd:    c7 44 24 04 cd 84 04        movl   $0x80484cd,0x4(%esp)
```

```
 80483e4:    08
 80483e5:    c7 04 24 d3 84 04 08      movl   $0x80484d3,(%esp)
 80483ec:    e8 03 ff ff ff            call   80482f4 <printf@plt>
        return 0;
 80483f1:    b8 00 00 00 00            mov    $0x0,%eax
}
```

看来真的是gcc做了优化。

**秘密五：链接了哪些共享库**

用ldd查看helloworld链接的共享库。

屏幕打印

```
linux-gate.so.1 =>  (0x002da000)
libc.so.6 => /lib/libc.so.6 (0x006e6000)
/lib/ld-linux.so.2 (0x006c6000)
```

libc.so.6是glibc，它实现了像printf这类标准C的函数。

ld-linux.so.2是ELF可执行文件的解释器，Linux内核在执行ELF可执行文件时，其实是执行ld-linux.so.2，然后由ld-linux.so.2去加载可执行文件及其所依赖的共享库。/lib/ld-linux.so.2是共享库，但它又是可执行的，运行它，屏幕会打印以下内容。

```
Usage: ld.so [OPTION]... EXECUTABLE-FILE [ARGS-FOR-PROGRAM...]
You have invoked `ld.so', the helper program for shared library executables.
This program usually lives in the file `/lib/ld.so', and special directives
in executable files using ELF shared libraries tell the system's program
loader to load the helper program from this file.  This helper program loads
the shared libraries needed by the program executable, prepares the program
to run, and runs it.  You may invoke this helper program directly from the
command line to load and run an ELF executable file; this is like executing
that file itself, but always uses this helper program from the file you
specified, instead of the helper program file specified in the executable
file you run.  This is mostly of use for maintainers to test new versions
of this helper program; chances are you did not intend to run this program.

      --list                list all dependencies and how they are resolved
      --verify              verify that given object really is a dynamically linked
                            object we can handle
      --library-path PATH   use given PATH instead of content of the environment
                            variable LD_LIBRARY_PATH
      --inhibit-rpath LIST  ignore RUNPATH and RPATH information in object names
                            in LIST
```

从ld-linux.so.2 的参数来看，它可以直接执行ELF文件:

```
/lib/ld-linux.so.2 ./helloworld_5
```

屏幕打印

```
Hello World!
```

这和直接执行./helloworld_5的效果一样。

linux-gate.so.1则是有点奇怪了，它没有指向任何文件，而是指向一个地址0×002da000。这个文件又称为虚拟动态共享库（VDSO），linux- gate.so.1文件是不存在的，Linux内核根据CPU类型，动态决定使用哪个共享库。它的功能主要是加速系统调用（syscall）。

我们知道，用户空间代码（如应用程序）和内核代码是运行在不同级别的，用户空间代码的运行权限较小，内核代码的运行权限最高。由用户空间进入内核空间，需要跨越一个门（gate），这里linux-gate.so.1的功能就是提供这样一个门。

系统调用需要跨越用户空间与内核空间之间的门。在x86系列CPU上，Linux传统的做法是使用80中断（int 0x80）实现系统调用，不过它的执行效率较低。有的CPU提供了高效的sysenter指令，但不是所有CPU都支持，Linux通过VDSO来兼容这两种系统调用。于是提供了两个共享库。

```
ls -l /lib/modules/$(uname -r)/vdso

-rwxr-xr-x 1 root root 1764 03-24 12:10 vdso32-int80.so
-rwxr-xr-x 1 root root 1784 03-24 12:10 vdso32-sysenter.so
```

我们看sysenter的实现方式。

```
objdump -S vdso32-sysenter.so

00000400 <__kernel_sigreturn>:
 400:    58                          pop    %eax
 401:    b8 77 00 00 00              mov    $0x77,%eax
 406:    cd 80                       int    $0x80
 408:    90                          nop
 409:    8d 76 00                    lea    0x0(%esi),%esi

0000040c <__kernel_rt_sigreturn>:
 40c:    b8 ad 00 00 00              mov    $0xad,%eax
 411:    cd 80                       int    $0x80
 413:    90                          nop

00000414 <__kernel_vsyscall>:
 414:    51                          push   %ecx
 415:    52                          push   %edx
 416:    55                          push   %ebp
 417:    89 e5                       mov    %esp,%ebp
 419:    0f 34                       sysenter
 41b:    90                          nop
 41c:    90                          nop
 41d:    90                          nop
```

```
 41e:    90                      nop
 41f:    90                      nop
 420:    90                      nop
 421:    90                      nop
 422:    eb f3                   jmp    417 <__kernel_vsyscall+0x3>
 424:    5d                      pop    %ebp
 425:    5a                      pop    %edx
 426:    59                      pop    %ecx
 427:    c3                      ret
```

里面实现了kernel_sigreturn、kernel_rt_sigreturn和kernel_vsyscall。

vdso32-int80.so也实现了这几个函数，只是方法不一样。

```
00000400 <__kernel_sigreturn>:
 400:    58                      pop    %eax
 401:    b8 77 00 00 00          mov    $0x77,%eax
 406:    cd 80                   int    $0x80
 408:    90                      nop
 409:    8d 76 00                lea    0x0(%esi),%esi

0000040c <__kernel_rt_sigreturn>:
 40c:    b8 ad 00 00 00          mov    $0xad,%eax
 411:    cd 80                   int    $0x80
 413:    90                      nop

00000414 <__kernel_vsyscall>:
 414:    cd 80                   int    $0x80
 416:    c3                      ret
```

**秘密六：调用共享库中的函数**

puts是libc提供的函数，从反汇编代码中可以看到。

```
    printf("Hello World!\n");
80483c5:    c7 04 24 a4 84 04 08 movl   $0x80484a4,(%esp)
80483cc:    e8 1f ff ff ff          call   80482f0 <puts@plt>
```

从前面的分析中，我们已经知道，gcc的优化把printf换成了puts。但是这里也并没有直接调用puts，而是调用的puts@plt，这是怎么回事呢？puts@plt显然是编译器加的一个中间函数，我们看一下这个函数对应的汇编代码。

```
080482f0 <puts@plt>:
 80482f0:    ff 25 1c 96 04 08       jmp    *0x804961c
 80482f6:    68 10 00 00 00          push   $0x10
 80482fb:    e9 c0 ff ff ff          jmp    80482c0 <_init+0x30>
```

现在我们用调试器来对其进行分析。

```
gdb helloworld
```

```
(gdb) b main
Breakpoint 1 at 0x80483c5: file helloworld.c, line 5.
(gdb) r
Starting program: /home/work/mine/sysprog/think-in-compway/helloworld/helloworld

Breakpoint 1, main () at helloworld.c:5
5        printf("Hello World!\n");
Missing separate debuginfos, use: debuginfo-install glibc.i686
```

puts@plt先跳到*0x804961c，我们看看*0x804961c 里有什么？

```
(gdb) x 0x804961c
0x804961c <_GLOBAL_OFFSET_TABLE_+20>:    0x080482f6
```

*0x804961c等于0x080482f6，这正是puts@plt中的第二行汇编代码的地址。也就是说puts@plt整个函数会顺序执行，直到跳转到0x80482c0。

再来看看0x80482c0处有什么，通过汇编可以看到。

```
 080482c0 <__gmon_start__@plt-0x10>:
  80482c0:   ff 35 0c 96 04 08       pushl  0x804960c
  80482c6:   ff 25 10 96 04 08       jmp    *0x8049610
```

又跳到了*0x8049610，转的弯真多，没关系，我们再看*0x8049610。

```
(gdb) x 0x8049610
0x8049610 <_GLOBAL_OFFSET_TABLE_+8>:0x006da4d0
(gdb) x /wa 0x006da4d0
0x6da4d0 <_dl_runtime_resolve>:0x8b525150
```

原来转来转去就是为了调用函数_dl_runtime_resolve，_dl_runtime_resolve的功能就是找到要调用函数puts的地址。

为什么不直接调用_dl_runtime_resolve，而要转这么多圈子呢？

先执行完puts。

```
(gdb) n
```

再回头来看看puts@plt的第一行代码。

```
80482f0:   ff 25 1c 96 04 08       jmp    *0x804961c
```

通过调试器进行分析：

```
(gdb) x 0x804961c
0x804961c <_GLOBAL_OFFSET_TABLE_+20>:0x745af0 <puts>
```

对比前面的代码：

```
(gdb) x 0x804961c
0x804961c <_GLOBAL_OFFSET_TABLE_+20>:    0x080482f6
```

这其实表明在第一次执行时，会通过_dl_runtime_resolve解析到函数地址，并将puts的地址保存到0x804962c里，这样以后执行时就直接调用了。

**秘密七：函数的解析过程**

LD_DEBUG是一个很有用的环境变量，通过它，我们可以对helloworld做更深入的分析，按下列方式运行helloworld。

```
LD_DEBUG=symbols ./helloworld
```

屏幕会打印如下结果。

```
...
      7264: symbol=malloc;  lookup in file=./helloworld [0]
      7264: symbol=malloc;  lookup in file=/lib/libc.so.6 [0]
      7264: symbol=calloc;  lookup in file=./helloworld [0]
      7264: symbol=calloc;  lookup in file=/lib/libc.so.6 [0]
      7264: symbol=realloc;  lookup in file=./helloworld [0]
      7264: symbol=realloc;  lookup in file=/lib/libc.so.6 [0]
      7264: symbol=free;  lookup in file=./helloworld [0]
      7264: symbol=free;  lookup in file=/lib/libc.so.6 [0]
      7264: symbol=puts;  lookup in file=./helloworld [0]
      7264: symbol=puts;  lookup in file=/lib/libc.so.6 [0]
...
```

查找函数时，先在可执行文件中查找，然后依次到共享库中查找。使用共享库，可以节省空间，但调用共享库的函数需要额外的开销。我们用静态链接的方式链接helloworld。

```
gcc -g -static helloworld.c -o helloworld_static
```

这样运行速度可能会快些，但是可执行文件的大小增加了不少。

```
ls -l helloworld_static helloworld
```

屏幕打印如下内容。

```
-rwxrwxr-x 1 root root   6128 05-16 17:40 helloworld
-rwxrwxr-x 1 root root 562578 05-16 17:40 helloworld_static
```

共享库的好处其实取决于共享的次数，共享的次数越多，因共享而节省的空间越多。如果一个函数库只有一个程序使用它，把它编译成静态库将是更好的选择。

**秘密八：托梁换柱**

下面的helloworld会在屏幕上打印出什么内容呢？

```
#include <stdio.h>

int main(int argc, char* argv[], char* env[])
{
        printf("Hello World!\n");

        return 0;
}
```

肯定是“Hello World!”，不是吗？下面我们来个托梁换柱。

```
/* preload.c */

#include <stdio.h>
#include <ctype.h>

int puts(const char *s)
{
        const char* p = s;
        while(*p != '\0')
        {
            putc(toupper(*p), stdout);
            p++;
        }

        return 0;
}
```

先编译该文件。

```
gcc -g -shared preload.c -o libpreload.so
```

再按下列方式运行helloworld。

```
LD_PRELOAD=./libpreload.so ./helloworld
```

屏幕会打印出如下内容。

```
HELLO WORLD!
```

设置环境变量LD_PRELOAD之后，打印的内容变成大写了！为什么呢？原来，LD_PRELOAD指定的共享库被预先加载，如果出现重名的函数，预先加载的函数将会被调用。通过这种方法，我们可以在不需要修改源代码（有时候可能没有源代码）的情况下，来改变一个程序的行为。

**秘密九：内存模型**

我们先对helloworld做点修改。

```
int main(int argc, char* argv[], char* env[])
{
```

```
        printf("Hello World!\n");

        getchar();

        return 0;
}
```

这里调用了getchar，让程序不会直接退出。利用程序等待输入的时间，我们看下它的内存布局（假设3458 是helloworld的进程ID）。

```
cat /proc/3458/maps

0041f000-00420000 r-xp 0041f000 00:00 0          [vdso]
006c6000-006e2000 r-xp 00000000 08:01 765655     /lib/ld-2.8.so
006e2000-006e3000 r--p 0001c000 08:01 765655     /lib/ld-2.8.so
006e3000-006e4000 rw-p 0001d000 08:01 765655     /lib/ld-2.8.so
006e6000-00849000 r-xp 00000000 08:01 765657     /lib/libc-2.8.so
00849000-0084b000 r--p 00163000 08:01 765657     /lib/libc-2.8.so
0084b000-0084c000 rw-p 00165000 08:01 765657     /lib/libc-2.8.so
0084c000-0084f000 rw-p 0084c000 00:00 0
08048000-08049000 r-xp 00000000 08:05 2129380
    /home/work/mine/sysprog/think-in-compway/helloworld/helloworld_9
08049000-0804a000 rw-p 00000000 08:05 2129380
/home/work/mine/sysprog/think-in-compway/helloworld/helloworld_9
b7f87000-b7f89000 rw-p b7f87000 00:00 0
b7fa6000-b7fa8000 rw-p b7fa6000 00:00 0
bfc93000-bfca8000 rw-p bffeb000 00:00 0          [stack]
```

从这里我们可以看出以下几点。

(1) 这里最低有效地址是0x0041f000，其下为保留区域。

(2) 可执行文件和共享库映射区，每个文件通常占1—3个区域，分别存放全局变量、常量和代码，它们有不同的属性。

(3) 这里没有动态分配内存，所以没有堆内存。

(4) 栈是从上向下增长的，这里栈底为0xbfca8000，我做了多次测试，发现栈底并不总是在这个位置。不过对于32位系统，我们可以确信的是栈底总是小于0xc0000000的。



**秘密十：main函数不是第一个执行的函数**

教科书告诉我们main函数是C语言程序的入口函数，实际上main并不是第一个被执行的函数。

我们对程序做点修改。

```
#include <stdio.h>

__attribute ((constructor)) void hello_init(void)
{
```

```
        printf("%s\n", __func__);

        return;
}

__attribute ((destructor)) void hello_fini(void)
{
        printf("%s\n", __func__);

        return;
}

int main(int argc, char* argv[], char* env[])
{
        printf("Hello World!\n");

        return 0;
}
```

编译并运行：

```
./helloworld_10
```

屏幕打印：

```
hello_init
Hello World!
hello_fini
```

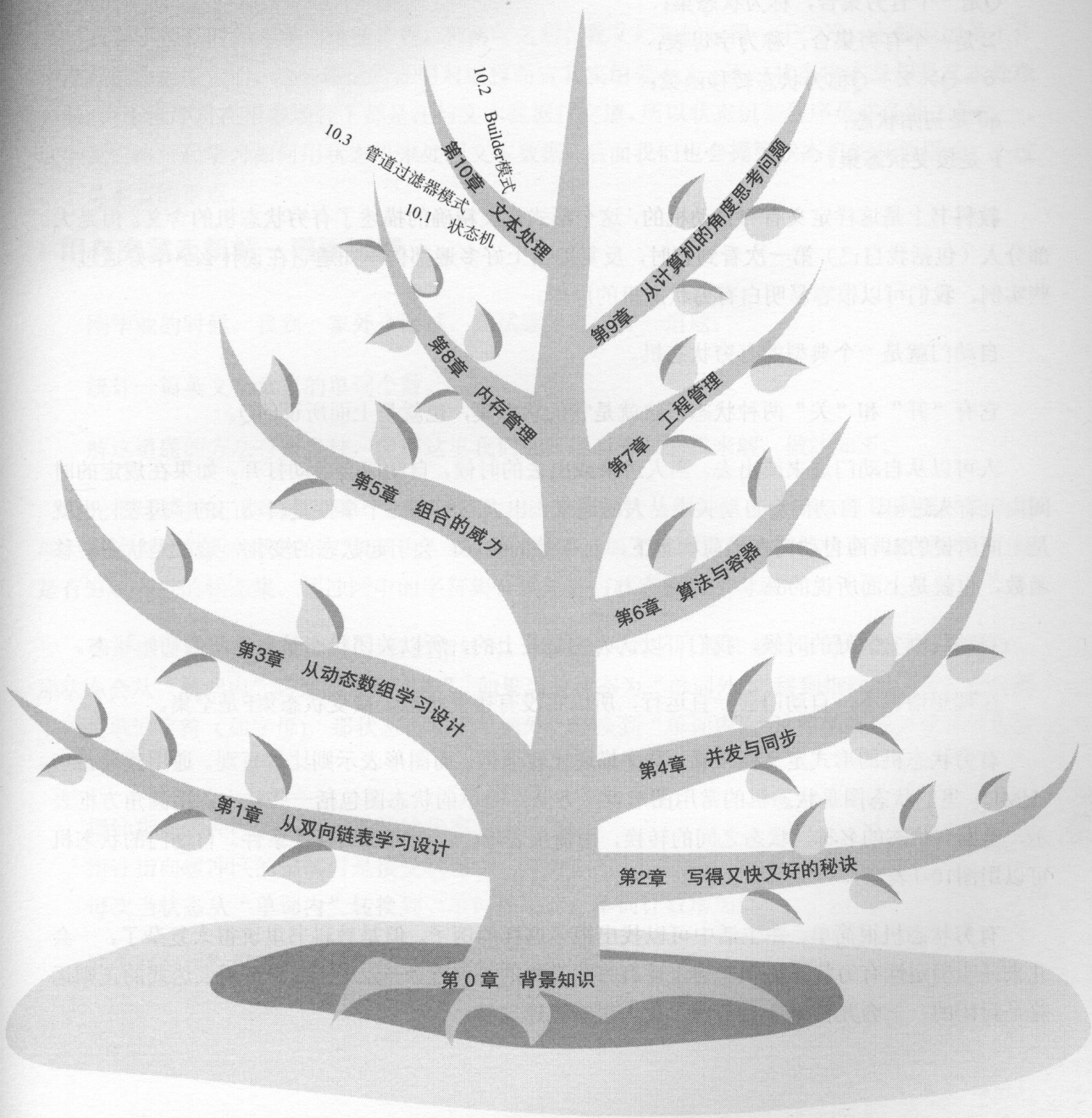
在main函数之前执行了hello_init ，在main函数之后执行了hello_fini。在gdb里执行./helloworld_10，并在hello_init 和hello_fini设置断点，看是谁调用了它们。

```
(gdb) bt
#0  hello_init () at helloworld.10.c:5
#1  0x0804849d in __do_global_ctors_aux ()
#2  0x080482bc in _init ()
#3  0x08048439 in __libc_csu_init ()
#4  0x006fc571 in __libc_start_main () from /lib/libc.so.6
#5  0x08048321 in _start ()

(gdb) bt
#0  hello_fini () at helloworld.10.c:12
#1  0x0804836f in __do_global_dtors_aux ()
#2  0x080484c4 in _fini ()
#3  0x006d4f7b in _dl_fini () from /lib/ld-linux.so.2
#4  0x00713b39 in exit () from /lib/libc.so.6
#5  0x006fc5de in __libc_start_main () from /lib/libc.so.6
#6  0x08048321 in _start ()
```

其实_start才是程序的入口，它先构造进程中的全局对象，执行一些初始化函数，然后调用main函数，最后析构全局对象，执行一些退出函数。

# 第 10 章

# 文本处理


10.2 Builder模式
10.3 管道过滤器模式
10.1 状态机
第10章 文本处理
第9章 从计算机的角度思考问题
第8章 内存管理
第7章 工程管理
第5章 组合的威力
第6章 算法与容器
第3章 从动态数组学习设计
第4章 并发与同步
第1章 从双向链表学习设计
第2章 写得又快又好的秘诀
第 0 章 背景知识

# 10.1 状态机

## 有穷状态机的定义

有穷状态机是一个五元组(Q，Σ，δ，q0，F)，其中：

Q是一个有穷集合，称为状态集；

Σ是一个有穷集合，称为字母表；

δ＝Q×Σ→Q称为状态转移函数；

q0 是初始状态；

F 是接受状态集。

教科书上是这样定义有穷自动机的，这个形式定义精确的描述了有穷状态机的含义。但是大部分人（包括我自己）第一次看到它时，反复地读上好多遍都仍不知道它在说什么。幸好通过一些实例，我们可以很容易明白有穷状态机的原理。

自动门就是一个典型的有穷状态机。

它有“开”和“关”两种状态，这就是它的状态集，也就是上面所说的Q。

人可以从自动门进来或出去，当人进来或出去的时候，自动门会自动打开，如果在规定的时间内没有人进出，自动门会自动关上。人的进来、出去和超时三个事件是自动门的字母表，也就是上面所说的Σ。而自动门在当前状态下，对事件的响应，会引起状态的变化，这就是状态转移函数，也就是上面所说的δ。

自动门刚安装好的时候，我们可以认为它是关上的，所以关闭状态是自动门的初始状态。

在理想情况下，自动门会一直运行，所以它没有接受状态，接受状态集F是空集。

有穷状态机的形式定义很精确，文字描述比较通俗，而图形表示则比较直观。通用建模语言（UML）里的状态图是状态机的常用图形表示方法。简单的状态图包括一些状态，用圆角方框表示，里面有状态的名称。状态之间的转换，用箭头表示，上面可以加转换条件。自动门的状态机可以用图10-1表示。

有穷状态机很简单，在生活中可以找出很多这样的例子。但是教科书里讲得太复杂了，一会儿来证明确定性有穷状态机和非确定性有穷状态机的等价性，一会儿又证明正则表达式的正则运算是封闭的，一会儿再来个泵引理，让人很难迅速理解。

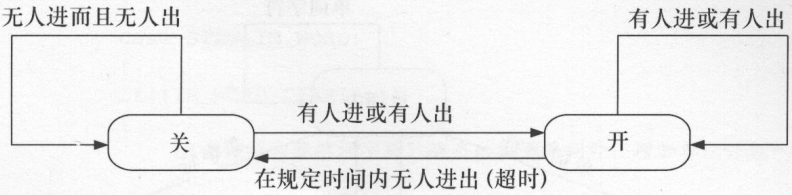



图10-1 自动门的状态机

我花了很长时间才明白这些原理，但两年之后，我又把它们忘得一干二净。主要原因是工作中没有机会运用它们，这些理论的证明对编程而言其实用处不大，不过状态机本身却是文本处理利器，由于程序员在很多场合下都是在与文本数据打交道，所以状态机是程序员必备的工具之一。这里我们将一起学习如何用状态机来处理文本数据，后面我们也会提到状态机的其他用途，不过这不是本节的重点。

## 用有穷状态机解一道面试题

刚毕业的时候，我到一家外企面试，面试题里有这样一道题：

统计一篇英文文章里的单词个数。

解这道题的方法有很多种，但在这里我们选择用有穷状态机来解，做法如下。

先把这篇英文文章读入到一个缓冲区里，让一个指针从缓冲区的头部一直移到缓冲区的尾部，指针会处于两种状态：“单词内”或“单词外”，加上后面提到的初始状态和接受状态，就是有穷状态机的状态集。缓冲区中的字符集合就是有穷状态机的字母表。

如果当前状态为“单词内”，移到指针时，指针指向的字符是非单词字符（如标点和空格），那状态会从“单词内”转换到“单词外”。如果当前状态为“单词外”，移到指针时，指针指向的字符是单词字符（如字母），那状态会从“单词外”转换到“单词内”。这些转换规则就是状态转换函数。

指针指向缓冲区的头部时是初始状态。
指针指向缓冲区的尾部时是接受状态。
每次当状态从“单词内”转换到“单词外”时，单词计数增加1。

这个有穷状态机的图形如图10-2所示。

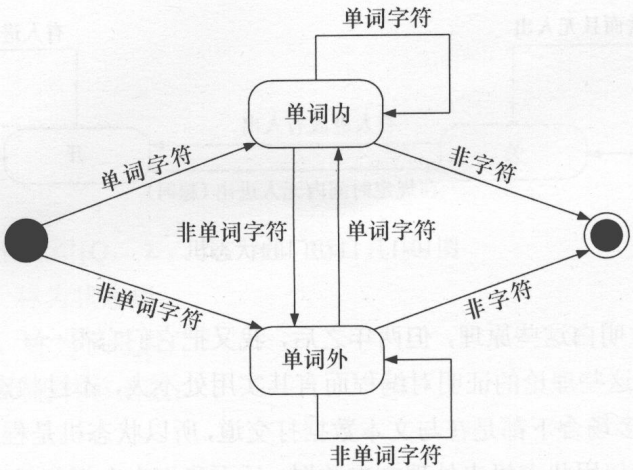



图10-2　统计英文文章中单词个数的有穷状态机

下面我们看看这个程序具体该怎么写。

```
int count_word(const char* text)
{
    /*定义各种状态，我们不关心接受状态，这里可以不用定义。*/
        enum _State
        {
            STAT_INIT,
            STAT_IN_WORD,
            STAT_OUT_WORD,
        }state = STAT_INIT;

        int count = 0;
        const char* p = text;

        /*在一个循环中，指针从缓冲区头移动缓冲区尾*/
        for(p = text; *p != '\0'; p++)
        {
            switch(state)
            {
                case STAT_INIT:
                {
                    if(IS_WORD_CHAR(*p))
                    {
                        /*指针指向单词字符，状态转换为单词内*/
                        state = STAT_IN_WORD;
                    }
                    else
                    {
                        /*指针指向非单词字符，状态转换为单词外*/
                        state = STAT_OUT_WORD;
                    }
                    break;
```

```
                    }
                    case STAT_IN_WORD:
                    {
                    if(!IS_WORD_CHAR(*p))
                    {
                        /*指针指向非单词字符，状态转换为单词外，增加单词计数*/
                        count++;
                        state = STAT_OUT_WORD;
                    }
                    break;
                    }
                    case STAT_OUT_WORD:
                    {
                    if(IS_WORD_CHAR(*p))
                    {
                        /*指针指向单词字符，状态转换为单词内*/
                        state = STAT_IN_WORD;
                    }
                    break;
                    }
                    default:break;
            }
        }

        if(state == STAT_IN_WORD)
    {
            /*如果由单词内进入接受状态，增加单词计数*/
            count++;
        }

        return count;
}
```

用状态机来解这道题目，思路清晰，程序简单，不易出错。

这道题目只是为了展示一些奇技淫巧，还是有一些实际用处呢？在回答这个问题之前，我们先对上面的程序做点扩展，让它不只能统计单词的个数，而且要分离出里面的每个单词。

```
int word_segmentation(const char* text, OnWordFunc on_word, void* ctx)
{
        enum _State
        {
            STAT_INIT,
            STAT_IN_WORD,
            STAT_OUT_WORD,
        }state = STAT_INIT;

        int count = 0;
        char* copy_text = strdup(text);
        char* p = copy_text;
        char* word = copy_text;
```

```
    for(p = copy_text; *p != '\0'; p++)
    {
        switch(state)
        {
            case STAT_INIT:
            {
            if(IS_WORD_CHAR(*p))
            {
                word = p;
                state = STAT_IN_WORD;
            }
            break;
            }
            case STAT_IN_WORD:
            {
            if(!IS_WORD_CHAR(*p))
            {
                count++;
                *p = '\0';
                on_word(ctx, word);
                state = STAT_OUT_WORD;
            }
            break;
            }
            case STAT_OUT_WORD:
            {
            if(IS_WORD_CHAR(*p))
            {
                word = p;
                state = STAT_IN_WORD;
            }
            break;
            }
            default:break;
        }
    }

    if(state == STAT_IN_WORD)
    {
        count++;
        on_word(ctx, word);
    }

    free(copy_text);

    return count;
}
```

状态机不变，只是在状态转换时，做的事情不一样。这里从“单词内”转换到其他状态时，增加单词计数，并分离出当前的单词。至于拿分离出的单词来做什么，由传入的回调函数决定，比如可以用来统计每个单词出现的频率。

但如果讨论还是限于英文文章，这个程序的意义仍然不大，现在来做进一步扩展。我们考虑的文本不再是英文文章，而是一些文本数据，这些数据由一些分隔符分开，我们把数据称为token，现在我们要把这些token分离出来。

```
typedef void (*OnTokenFunc)(void* ctx, int index, const char* token);

#define IS_DELIM(c) (strchr(delims, c) != NULL)
int parse_token(const char* text, const char* delims, OnTokenFunc on_token, void* ctx)
{
        enum _State
        {
            STAT_INIT,
            STAT_IN,
            STAT_OUT,
        }state = STAT_INIT;

        int   count     = 0;
        char* copy_text = strdup(text);
        char* p         = copy_text;
        char* token     = copy_text;

        for(p = copy_text; *p != '\0'; p++)
        {
            switch(state)
            {
                case STAT_INIT:
                case STAT_OUT:
                {
                if(!IS_DELIM(*p))
                {
                    token = p;
                    state = STAT_IN;
                }
                break;
                }
                case STAT_IN:
                {
                if(IS_DELIM(*p))
                {
                    *p = '\0';
                    on_token(ctx, count++, token);
                    state = STAT_OUT;
                }
                break;
                }
                default:break;
            }
        }

        if(state == STAT_IN)
        {
            on_token(ctx, count++, token);
```

```
        }
        on_token(ctx, -1, NULL);
    free(copy_text);

    return count;
}
```

用分隔符分隔的文本数据有很多种，例如以下几种。

(1) 环境PATH，它由以“:”分开的多个路径组成。

```
/usr/lib/qt-3.3/bin:/usr/kerberos/bin:/backup/tools/jdk1.5.0_18/bin/:/usr/lib/ccache:/usr/local/bin:/bin:/usr/bin:/home/lixianjing/bin
```

(2) 文件名，它由以“/”分开的路径组成。

```
/usr/lib/qt-3.3/bin
```

(3) URL中的参数，它由以“&”分开的多个“键/值”对组成。

```
hl=zh-CN&q=limodev&btnG=Google+搜索&meta=&aq=f&oq=
```

所有这些数据都可以用上面的函数处理，所以这个小函数是颇具实用价值的。

## INI解析器

上面我们看了只有中间两个状态的状态机，现在我们来看一个稍微复杂一点的状态机。

INI文件是Windows下常用的一种配置文件。它由多个分组组成，每个组有多个配置项，每个配置项又由名称和值组成。文件里还可以包含注释，注释通常以“;”(或“#”)开始，直到当前行结束。如XP下的win.ini文件。

```
; for 16-bit app support
[fonts]
[extensions]
[mci extensions]
[files]
[MCI Extensions.BAK]
aif=MPEGVideo
aifc=MPEGVideo
aiff=MPEGVideo
asf=MPEGVideo
asx=MPEGVideo
au=MPEGVideo
m1v=MPEGVideo
m3u=MPEGVideo
mp2=MPEGVideo
mp2v=MPEGVideo
```

```
mp3=MPEGVideo
[annie]
CaptureFile=
VideoDevice=0
AudioDevice=0
FrameRate=333333
UseFrameRate=1
CaptureAudio=1
WantPreview=1
MasterStream=-1
[SciCalc]
layout=0
```

第一行是注释，后面有fonts、extensions和mci extensions三个空的分组，而MCI Extensions.BAK、annie和SciCalc三个分组包含有一个或多个配置项。

对于这样一个文件，我们应该怎样去解析它呢？按照前面的方法，先把数据读入到一个缓冲区中，让一个指针指向缓冲区的头部，然后移动指针，直到指向缓冲区的尾部。在这个过程中，指针可能指向的注释、分组的组名、配置项的名称、配置项的值或者一些如换行符之类的格式信息。

由此，我们可以这样来定义INI的状态机。

状态集合如下：

(1) 分组的组名状态；
(2) 注释状态；
(3) 配置项的名称状态；
(4) 配置项的值状态；
(5) 空白状态。

状态转换函数如下：

(1) 初始状态为“空白”状态；
(2) 在“空白”状态下，读入字符“[”，进入“分组组名”状态；
(3) 在“分组组名”状态下，读入字符“]”，分组组名解析成功，回到“空白”状态；
(4) 在“空白”状态下，读入字符“;”，进入“注释”状态；
(5) 在“注释”状态下，读入换行字符，结束“注释”状态，回到“空白”状态；
(6) 在“空白”状态下，读入非空白字符，进入“配置项的名称”状态；
(7) 在“配置项的名称”状态下，读入字符“=”，配置项的名称解析成功，进入“配置项的值”状态；
(8) 在“配置项的值”状态下，读入换行字符，配置项的值解析成功，回到“空白”状态。

INI状态机可以用图10-3来表示。

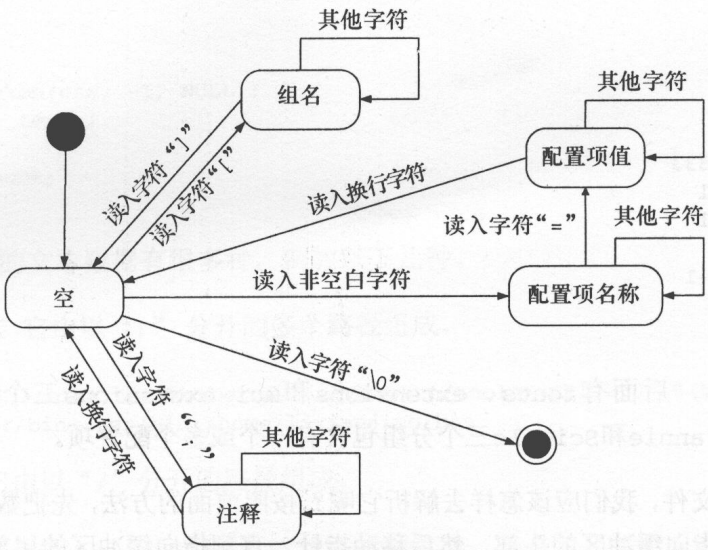



图10-3　INI状态机

现在我们来看看程序的具体实现。

```
static void ini_parse (char* buffer, char comment_char, char delim_char)
{
        char* p = buffer;
        char* group_start = NULL;
        char* key_start   = NULL;
        char* value_start = NULL;
        /*定义INI解析器的状态，初始状态为“空白”状态。*/
        enum _State
        {
            STAT_NONE = 0,
            STAT_GROUP,
            STAT_KEY,
            STAT_VALUE,
            STAT_COMMENT
        }state = STAT_NONE;

        for(p = buffer; *p != '\0'; p++)
        {
            switch(state)
            {
                                case STAT_NONE:
                                {
                if(*p == '[')
                {
                    /*在“空白”状态下，读入字符“[”，进入“分组组名”状态。*/
                    state = STAT_GROUP;
                    group_start = p + 1;
                }
```

```
else if(*p == comment_char)
{
    /*在“空白”状态下，读入字符“;”，进入“注释”状态。*/
    state = STAT_COMMENT;
}
else if(!isspace(*p))
{
    /*在“空白”状态下，读入非空白字符，进入“配置项的名称”状态。*/
    state = STAT_KEY;
    key_start = p;
}
break;
}
case STAT_GROUP:
{
/*在“分组组名”状态下，读入字符“]”,分组组名解析成功，
回到“空白”状态。*/
if(*p == ']')
{
    *p = '\0';
    state = STAT_NONE;
    strtrim(group_start);
    printf("[%s]\n", group_start);
}
break;
}
case STAT_COMMENT:
{
/*在“注释”状态下，读入换行字符，结束“注释”状态，回到“空白”状态。*/
if(*p == '\n')
{
    state = STAT_NONE;
    break;
}
break;
}
case STAT_KEY:
{
/*在“配置项的名称”状态下，读入字符“=”，
配置项的名称解析成功，进入“配置项的值”状态。*/
if(*p == delim_char || (delim_char == ' ' && *p == '\t'))
{
    *p = '\0';
    state = STAT_VALUE;
    value_start = p + 1;
}
break;
}
case STAT_VALUE:
{
/*在“配置项的值”状态下，读入换行字符，
配置项的值解析成功，回到“空白”状态。*/
if(*p == '\n' || *p == '\r')
```

```
                {
                    *p = '\0';
                    state = STAT_NONE;
                    strtrim(key_start);
                    strtrim(value_start);
                    printf("%s%c%s\n", key_start, delim_char, value_start);
                }
                break;
                }
                default:break;
            }
        }

        if(state == STAT_VALUE)
        {
            strtrim(key_start);
            strtrim(value_start);
            printf("%s%c%s\n", key_start, delim_char, value_start);
        }

        return;
}
```

INI文件有以下几个变种。

(1) 支持默认分组，如果只有一个分组，省略分组的组名，Linux中不少配置文件采用了这种方式。

(2) 注释符号，有的用“;”，有的用“#”，前者多用于Windows，后面多用于Linux。

(3) 名称和值之间的分隔，有的用空格，有的用“=”，还有的用“:”。

不管哪种格式，其解析方法都是一样的。在上面的程序中，我们使用了comment_char和delim_char两个参数来分别表示注释符号和分隔符号。

## XML解析器

XML（Extensible Markup Language，可扩展标记语言）也是一种常用的数据文件格式。相对于INI来说，它要复杂得多，INI只能保存线性结构的数据，而XML可以保存树形结构的数据。先看下面的例子。

```
<?xml version="1.0" encoding="utf-8"?>
<mime-type xmlns="http://www.freedesktop.org/standards/shared-mime-info" type="all/all">
        <!--Created automatically by update-mime-database. DO NOT EDIT!-->
        <comment>all files and folders</comment>
</mime-type>
```

第一行称为处理指令（PI），是给解析器用的。这里告诉解析器，当前的XML文件遵循XML 1.0规范，文件内容用UTF-8编码。

第二行是一个起始标签，标签的名称为mime-type。它有两个属性，第一个属性的名称为xmlns，值为 http://www.freedesktop.org/standards/shared-mime-info。第二个属性的名称为type，值为 all/all。

第三行是一句注释。

第四行包括一个起始标签，一段文本和结束标签。

第五行是一个结束标签。

XML本身的格式不是本文的重点，我们不详细讨论了。这里的重点是如何用状态机解析格式复杂的数据。

按照前面的方法，先把数据读入到一个缓冲区中，让一个指针指向缓冲区的头部，然后移动指针，直到指向缓冲区的尾部。在这个过程中，指针可能指向起始标签、结束标签、注释、处理指令或文本。由此我们定义出状态机的主要状态：

(1) 起始标签状态；
(2) 结束标签状态；
(3) 注释状态；
(4) 处理指令状态；
(5) 文本状态。

由于起始标签、结束标签、注释和处理指令都在字符“<”和“>”之间，所以当读入字符“<”时，我们还无法知道当前的状态，为了便于处理，我们引入一个中间状态，称为“小于号之后”的状态。在读入字符“<”和“!”之后，还要读入两个“-”，才能确定进入注释状态，为了便于处理，再引入两个中间状态“注释前一”和“注释前二”。再引入一个“空”状态，表示不在上述任何状态中。

状态转换函数：

(1) 在“空”状态下，读入字符“<”，进入“小于号之后”状态；
(2) 在“空”状态下，读入非“<”非空白的字符，进入“文本”状态；
(3) 在“小于号之后”状态下，读入字符“!”，进入“注释前一”状态；
(4) 在“小于号之后”状态下，读入字符“?”，进入“处理指令”状态；
(5) 在“小于号之后”状态下，读入字符“/”，进入“结束标签”状态；
(6) 在“小于号之后”状态下，读入有效的ID字符，进入“起始标签”状态；
(7) 在“注释前一”状态下，读入字符“-”，进入“注释前二”状态；
(8) 在“注释前二”状态下，读入字符“-”，进入“注释”状态；
(9) 在“起始标签”状态、“结束标签”状态、“文本”状态、“注释”状态和“处理指令”状态结束后，重新回到“空”状态下。

这个状态机的图形表示如图10-4所示。

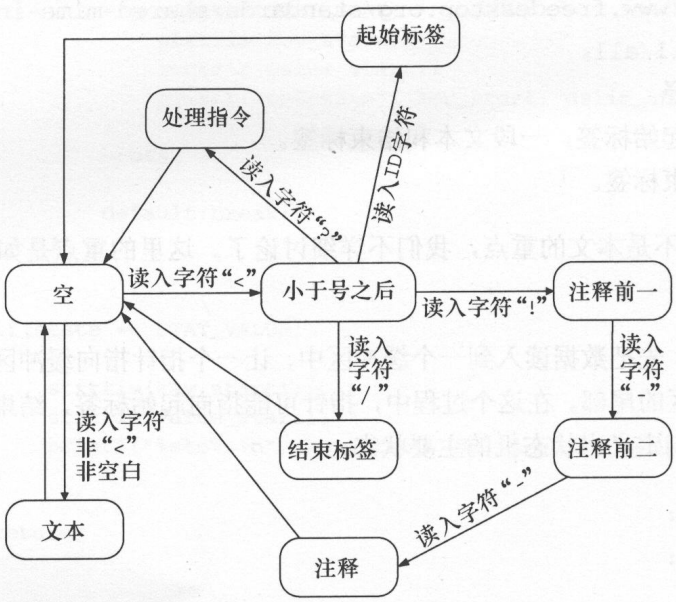



图10-4　简单XML解析器的状态机

下面我们来看看具体的代码实现。

```
void xml_parser_parse(XmlParser* thiz, const char* xml)
{
    /*定义状态的枚举值*/
    enum _State
    {
        STAT_NONE,
        STAT_AFTER_LT,
        STAT_START_TAG,
        STAT_END_TAG,
        STAT_TEXT,
        STAT_PRE_COMMENT1,
        STAT_PRE_COMMENT2,
        STAT_COMMENT,
        STAT_PROCESS_INSTRUCTION,
    }state = STAT_NONE;

    thiz->read_ptr = xml;
    /*指针从头移动到尾*/
    for(; *thiz->read_ptr != '\0'; thiz->read_ptr++)
    {
        char c = thiz->read_ptr[0];

        switch(state)
```

```
{
    case STAT_NONE:
    {
        if(c == '<')
        {
            /*在“空”状态下，读入字符“<”，进入“小于号之后”状态。*/
            xml_parser_reset_buffer(thiz);
            state = STAT_AFTER_LT;
        }
        else if(!isspace(c))
        {
            /*在“空”状态下，读入非“<”非空白的字符，进入“文本”状态。*/
            state = STAT_TEXT;
        }
        break;
    }
    case STAT_AFTER_LT:
    {
        if(c == '?')
        {
            /*在“小于号之后”状态下，读入字符“?”，进入“处理指令”状态。*/
            state = STAT_PROCESS_INSTRUCTION;
        }
        else if(c == '/')
        {
            /*在“小于号之后”状态下，读入字符“/”，进入“结束标签”状态。*/
            state = STAT_END_TAG;
        }
        else if(c == '!')
        {
            /*在“小于号之后”状态下，读入字符“!”，进入“注释前一”  状态*/
            state = STAT_PRE_COMMENT1;
        }
        else if(isalpha(c) || c == '_')
        {
            /*在“小于号之后”状态下，读入有效的ID字符，进入“起始标签”状态。*/
            state = STAT_START_TAG;
        }
        else
        {
        }
        break;
    }
    case STAT_START_TAG:
    {
        /*进入子状态*/
        xml_parser_parse_start_tag(thiz);
        state = STAT_NONE;
        break;
    }
    case STAT_END_TAG:
```

```
{
    /*进入子状态*/
    xml_parser_parse_end_tag(thiz);
    state = STAT_NONE;
    break;
}
case STAT_PROCESS_INSTRUCTION:
{
    /*进入子状态*/
    xml_parser_parse_pi(thiz);
    state = STAT_NONE;
    break;
}
case STAT_TEXT:
{
    /*进入子状态*/
    xml_parser_parse_text(thiz);
    state = STAT_NONE;
    break;
}
case STAT_PRE_COMMENT1:
{
    if(c == '-')
    {
        /*在“注释前一” 状态下，读入字符“-”， 进入“注释前二” 状态。*/
        state = STAT_PRE_COMMENT2;
    }
    else
    {
    }
    break;
}
case STAT_PRE_COMMENT2:
{
    if(c == '-')
    {
    /*在“注释前二” 状态下，读入字符“-”， 进入“注释” 状态。*/
        state = STAT_COMMENT;
    }
    else
    {
    }
}
case STAT_COMMENT:
{
    /*进入子状态*/
    xml_parser_parse_comment(thiz);
    state = STAT_NONE;
    break;
}
default:break;
```

```
            }
            if(*thiz->read_ptr == '\0')
            {
                break;
            }
        }
        return;
    }
```

解析并没有在此结束，原因是像“起始标签”状态和“处理指令”状态等，它们不是原子的，内部还包含一些子状态，如标签名称，属性名和属性值等，它们需要进一步分解。在考虑子状态时，我们可以忘掉它所处的上下文，只考虑子状态本身，这样问题会得到简化。下面看一下起始标签的状态机。

假设我们要解析下面这样一个起始标签。

```
<mime-type xmlns="http://www.freedesktop.org/standards/shared-mime-info" type="all/all">
```

我们应该怎样去做呢？还是按前面的方法，让一个指针指向缓冲区的头部，然后移动指针，直到指向缓冲区的尾部。在这个过程中，指针可能指向，标签名称，属性名和属性值。由此我们可以定义出状态机的主要状态：

(1) “标签名称”状态；
(2) “属性名”状态；
(3) “属性值”状态。

为了方便处理，再引两个中间状态，“属性名之前”状态和“属性值之前”状态。

状态转换函数如下（初始状态为“标签名称”状态）。

(1) 在“标签名称”状态下，读入空白字符，进入“属性名之前”状态。
(2) 在“标签名称”状态下，读入字符“/”或“>”，进入“结束”状态。
(3) 在“属性名之前”状态下，读入其他非空白字符，进入“属性名”状态。
(4) 在“属性名”状态下，读入字符“=”，进入“属性值之前”状态。
(5) 在“属性值之前”状态下，读入字符“ ‘ ”，进入“属性值”状态。
(6) 在“属性值”状态下，读入字符“ ’ ”，成功解析属性名和属性值，回到“属性名之前”状态。
(7) 在“属性名之前”状态下，读入字符“/”或“>”，进入“结束”状态。

由于处理指令（PI）里也包含了属性状态，为了重用属性解析的功能，我们把属性的状态再提取为一个子状态。这样，“起始标签”状态的图形表示如图10-5所示。

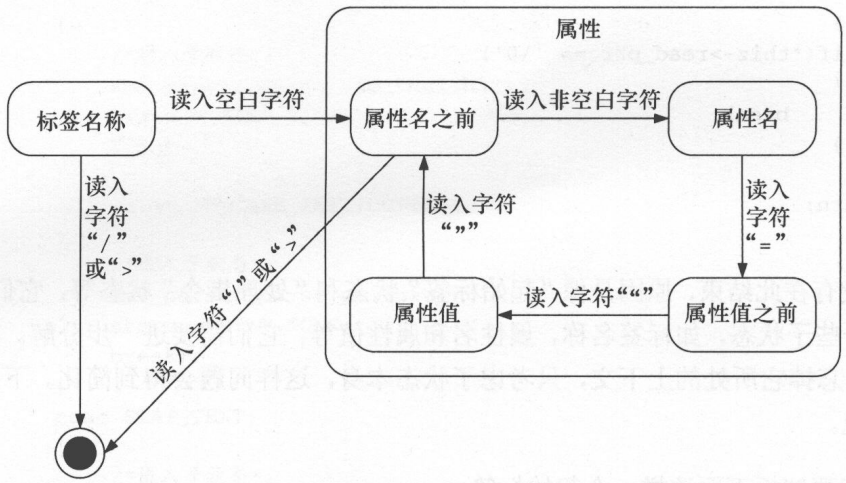



图10-5　“起始标签”状态机

下面我们看具体的代码实现。

```
static void xml_parser_parse_attrs(XmlParser* thiz, char end_char)
{
    int i = 0;
    enum _State
    {
        STAT_PRE_KEY,
        STAT_KEY,
        STAT_PRE_VALUE,
        STAT_VALUE,
        STAT_END,
    }state = STAT_PRE_KEY;

    char value_end = '\"';
    const char* start = thiz->read_ptr;

    thiz->attrs_nr = 0;
    for(; *thiz->read_ptr != '\0' && thiz->attrs_nr < MAX_ATTR_NR; thiz->read_ptr++)
    {
        char c = *thiz->read_ptr;

        switch(state)
        {
            case STAT_PRE_KEY:
            {
                if(c == end_char || c == '>')
                {
                        /*在“属性名之前”状态下，读入字符“/”或“>”，进入“结束”状态。*/
                        state = STAT_END;
                }
                else if(!isspace(c))
                {
                        /*在“属性名之前”状态下，读入其他非空白字符，进入“属性名”状态。*/
```

```
                state = STAT_KEY;
                start = thiz->read_ptr;
            }
        }
        case STAT_KEY:
        {
            if(c == '=')
            {
                    /*在“属性名”状态下，读入字符‘=’，进入“属性值之前”状态。*/
                    thiz->attrs[thiz->attrs_nr++] =
                    (char*)xml_parser_strdup(thiz, start, thiz->read_ptr - start);
                    state = STAT_PRE_VALUE;
            }

            break;
        }
        case STAT_PRE_VALUE:
        {
            /*在“属性值之前”状态下，读入字符“ ‘ ”，进入“属性值”状态。*/
            if(c == '\"' || c == '\'')
            {
                state = STAT_VALUE;
                value_end = c;
                start = thiz->read_ptr + 1;
            }
            break;
        }
        case STAT_VALUE:
        {
            /*在“属性值”状态下，读入字符“ ’ ”，成功解析属性名和属性值，
            回到“属性名之前”状态。*/
            if(c == value_end)
            {
                thiz->attrs[thiz->attrs_nr++] =
                (char*)xml_parser_strdup(thiz, start, thiz->read_ptr - start);
                state = STAT_PRE_KEY;
            }
        }
        default:break;
    }

    if(state == STAT_END)
    {
        break;
    }
}

for(i = 0; i < thiz->attrs_nr; i++)
{
    thiz->attrs[i] = thiz->buffer + (size_t)(thiz->attrs[i]);
}
thiz->attrs[thiz->attrs_nr] = NULL;

return;
}
```

因为在XML里单引号和双引号都可以用来界定属性值，所以上面对此做了特殊处理。

```
static void xml_parser_parse_start_tag(XmlParser* thiz)
{
    enum _State
    {
        STAT_NAME,
        STAT_ATTR,
        STAT_END,
    }state = STAT_NAME;

    char* tag_name = NULL;
    const char* start = thiz->read_ptr - 1;

    for(; *thiz->read_ptr != '\0'; thiz->read_ptr++)
    {
        char c = *thiz->read_ptr;

        switch(state)
        {
            case STAT_NAME:
            {
            /*在“标签名称”状态下，读入空白字符，进入“属性”子状态。*/
            /*在“标签名称”状态下，读入字符“/”或“>”，进入“结束”状态。*/
            if(isspace(c) || c == '>' || c == '/')
            {
                state = (c != '>' && c != '/') ? STAT_ATTR : STAT_END;
            }
            break;
        }
        case STAT_ATTR:
        {
                /*进入“属性”子状态*/
                xml_parser_parse_attrs(thiz, '/');
                state = STAT_END;

                break;
            }
            default:break;

        }

        if(state == STAT_END)
        {
            break;
        }
    }

    for(; *thiz->read_ptr != '>' && *thiz->read_ptr != '\0'; thiz->read_ptr++);

    return;
}
```

处理指令的解析和起始标签的解析基本上是一样的，这里只是看一下代码。

```
static void xml_parser_parse_pi(XmlParser* thiz)
{
    enum _State
    {
        STAT_NAME,
        STAT_ATTR,
        STAT_END
    }state = STAT_NAME;

    char* tag_name = NULL;
    const char* start = thiz->read_ptr;

    for(; *thiz->read_ptr != '\0'; thiz->read_ptr++)
    {
        char c = *thiz->read_ptr;

        switch(state)
        {
            case STAT_NAME:
            {
                /*在“标签名称”状态下，读入空白字符，进入“属性”子状态。*/
                /*在“标签名称”状态下，“>”，进入“结束”状态。*/
                if(isspace(c) || c == '>')
                {
                    state = c != '>' ? STAT_ATTR : STAT_END;
                }
                    break;
            }
            case STAT_ATTR:
            {
                /*进入“属性”子状态*/
                xml_parser_parse_attrs(thiz, '?');
                state = STAT_END;
                break;
            }
            default:break;
        }

        if(state == STAT_END)
        {
            break;
        }


    tag_name = thiz->buffer + (size_t)tag_name;

    for(; *thiz->read_ptr != '>' && *thiz->read_ptr != '\0'; thiz->read_ptr++);

    return;
}
```

注释、结束标签和文本的解析非常简单，这里结合代码看看就行了。

**“注释”子状态的处理**

```
static void xml_parser_parse_comment(XmlParser* thiz)
{
        enum _State
        {
            STAT_COMMENT,
            STAT_MINUS1,
            STAT_MINUS2,
        }state = STAT_COMMENT;

        const char* start = ++thiz->read_ptr;
        for(; *thiz->read_ptr != '\0'; thiz->read_ptr++)
        {
            char c = *thiz->read_ptr;

            switch(state)
            {
                case STAT_COMMENT:
                {
                /*在“注释”状态下，读入“-”，进入“减号一”状态。*/
                if(c == '-')
                {
                    state = STAT_MINUS1;
                }
                break;
                }
                case STAT_MINUS1:
                {
                if(c == '-')
                {
                    /*在“减号一”状态下，读入“-”，进入“减号二”状态。*/
                    state = STAT_MINUS2;
                }
                else
                {
                    state = STAT_COMMENT;
                }
                break;
                }
                case STAT_MINUS2:
                {
                if(c == '>')
                {
                    /*在“减号二”状态下，读入“>”，结束解析。*/
                    return;
                }
                else
                {
                    state = STAT_COMMENT;
                }
                }
```

```
                default:break;
            }
        }

        return;
}
```

**“结束标签”子状态的处理**

```
static void xml_parser_parse_end_tag(XmlParser* thiz)
{
        char* tag_name = NULL;
        const char* start = thiz->read_ptr;
        for(; *thiz->read_ptr != '\0'; thiz->read_ptr++)
{
        /*读入“>”，结束解析。*/
            if(*thiz->read_ptr == '>')
            {
                break;
            }
        }

        return;
}
```

**“文本”子状态的处理**

```
static void xml_parser_parse_text(XmlParser* thiz)
{
        const char* start = thiz->read_ptr - 1;
        for(; *thiz->read_ptr != '\0'; thiz->read_ptr++)
        {
            char c = *thiz->read_ptr;
            /*读入“>”，结束解析。*/
            if(c == '<')
            {
                if(thiz->read_ptr > start)
                {
                }
                thiz->read_ptr--;
                return;
            }
            else if(c == '&')
            {
            /*读入“&”，进入实体（entity）解析子状态。*/
                xml_parser_parse_entity(thiz);
            }
        }

        return;
}
```

实体（entity）子状态比较简单，这里就不做进一步分析了，留给读者做练习吧。

## 10.2 Builder模式

前面我们学习了状态机，并利用它来解析各种格式的文本数据。解析过程把线性的文本数据转换成一些基本的逻辑单元，但这通常只是任务的一部分，接下来我们还要对这些解析出来的数据做进一步处理。对于特定格式的文本数据，它的解析过程是一样的，但是对解析出来的数据而言，却有着多种多样的处理方式。为了让解析过程能被重用，就需要把数据的解析和数据的处理分离开来。

现在我们回过头来看一下前面写的函数parse_token，这个函数把用分隔符分隔的文本数据分离成一个一个的token。

parse_token的函数原型如下：

```
typedef void (*OnTokenFunc)(void* ctx, int index, const char* token);
int parse_token(const char* text, const char* delims, OnTokenFunc on_token, void* ctx)
```

parse_token负责解析数据，但它并不关心数据代表的意义及用途。对数据的进一步处理由调用者提供的回调函数来完成，函数 parse_token每解析到一个token，就调用这个回调函数。parse_token负责数据的解析，回调函数则负责数据的处理，这样一来，数据的解析和数据的处理就分开了。

我们可以认为parse_token是Builder模式最朴素的应用。现在我们看看Builder模式。

Builder 模式的意图：将一个复杂对象的构建与它的表示分离，使得同样的构建过程可以创建不同的表示。“构建”其实就是前面的解析过程，而“表示”就是前面说的对数据的处理。

对象关系如图10-6所示。

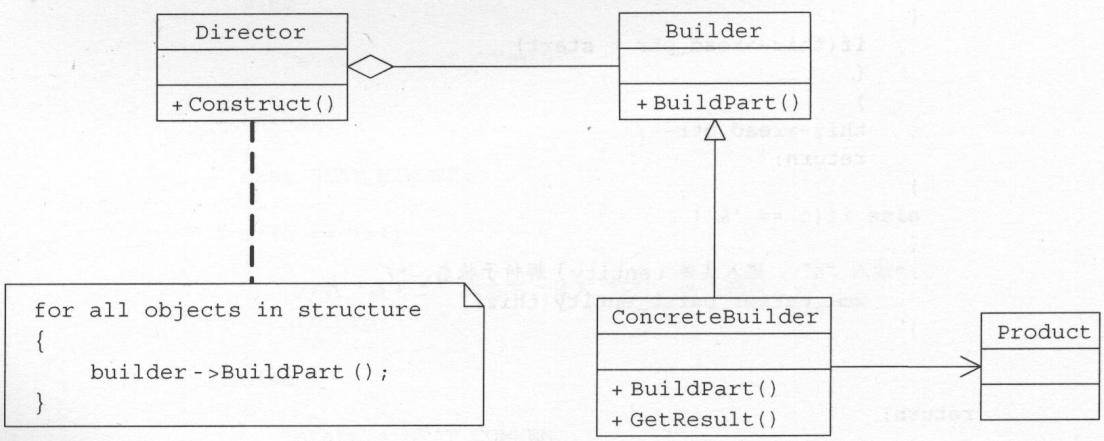



图10-6　Builder的对象关系图

前面提到的parse_token与这里的Director对应。

前面提到的回调函数与这里的Builder对应。

具体的回调函数与这里的ConcreteBuilder对应。

对数据处理的结果就是Product。

对象协作如图10-7所示。

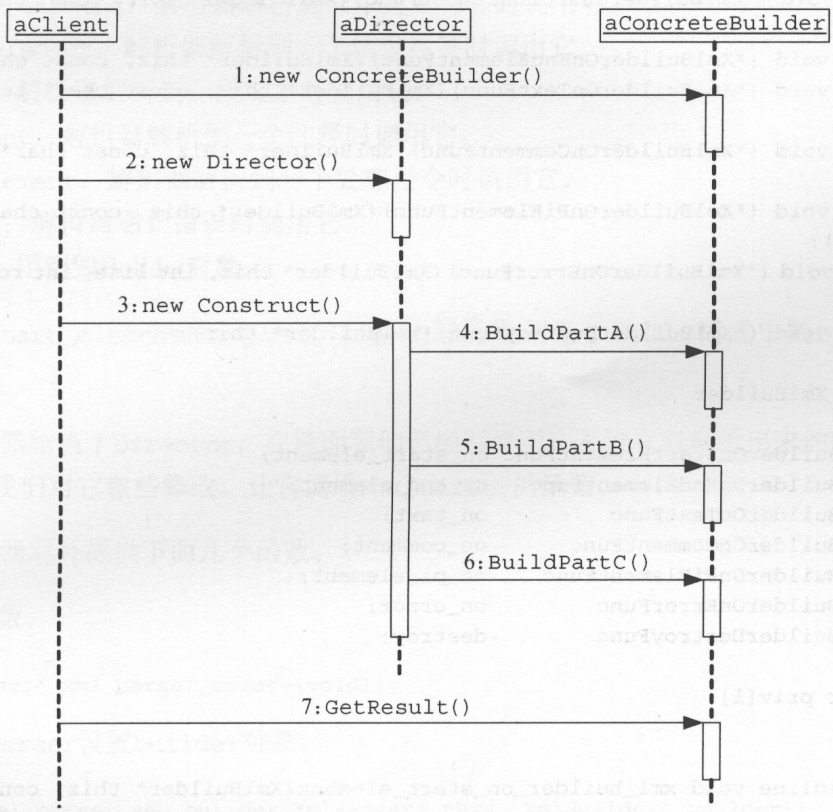



图10-7 Builder的对象协作图

Client是parse_token的调用者。

由于parse_token是按面向过程的方式设计的，所以ConcreteBuilder和Director的创建只是对应于一些初始化代码。

调用parse_token相当于调用aDirector的Construct函数。

调用回调函数相当于调用aConcreteBuilder的BuildPart函数。

回调函数可能把处理结果存在它的参数ctx中，GetResult是从里面获取结果，这是可选的过程，依赖于具体回调函数所做的工作。

parse_token的例子简单直接，对于理解Builder模式有较大的帮助，不过毕竟它是面向过程的。现在我们以前面的XML解析器为例来说明Builder模式，虽然我们的代码是用C写的，但完全是用面向对象的思想来设计的。Builder是一个接口，我们先把它定义出来。

```
struct _XmlBuilder;
typedef struct _XmlBuilder XmlBuilder;

typedef void (*XmlBuilderOnStartElementFunc)(XmlBuilder* thiz, const char* tag, const
char** attrs);
typedef void (*XmlBuilderOnEndElementFunc)(XmlBuilder* thiz, const char* tag);
typedef void (*XmlBuilderOnTextFunc)(XmlBuilder* thiz, const char* text, size_t
length);
typedef void (*XmlBuilderOnCommentFunc)(XmlBuilder* thiz, const char* text, size_t
length);
typedef void (*XmlBuilderOnPiElementFunc)(XmlBuilder* thiz, const char* tag, const
char** attrs);
typedef void (*XmlBuilderOnErrorFunc)(XmlBuilder* thiz, int line, int row, const char*
message);
typedef void (*XmlBuilderDestroyFunc)(XmlBuilder* thiz);

struct _XmlBuilder
{
    XmlBuilderOnStartElementFunc on_start_element;
    XmlBuilderOnEndElementFunc   on_end_element;
    XmlBuilderOnTextFunc         on_text;
    XmlBuilderOnCommentFunc      on_comment;
    XmlBuilderOnPiElementFunc    on_pi_element;
    XmlBuilderOnErrorFunc        on_error;
    XmlBuilderDestroyFunc        destroy;

    char priv[1];
};

static inline void xml_builder_on_start_element(XmlBuilder* thiz, const char* tag,
const char** attrs)
{
    return_if_fail(thiz != NULL && thiz->on_start_element != NULL);

    thiz->on_start_element(thiz, tag, attrs);

    return;
}

static inline void xml_builder_on_end_element(XmlBuilder* thiz, const char* tag)
{
    return_if_fail(thiz != NULL && thiz->on_end_element != NULL);

    thiz->on_end_element(thiz, tag);
```

```
    return;
}

...
```

（其他inline函数就不列在这里了。）

XmlBuilder接口要求实现下列函数。

on_start_element：解析器解析到一个起始标签时调用它。

on_end_element：解析器解析到一个结束标签时调用它。

on_text：解析器解析到一段文本时调用它。

on_comment：解析器解析到一个注释时调用它。

on_pi_element：解析器解析到一个处理指令时调用它。

on_error：解析器遇到错误时调用它。

destroy：用销毁Builder对象。

其中on_start_element和on_end_element等函数相当于Builder模式中的BuildPart*X*函数。

XML解析器相当于Director，在前面我们已经写好了，不过它对解析出来的数据没有做任何处理。现在我们对它做些修改，让它调用XmlBuilder的函数。

XML解析器对外提供下面几个函数。

- 构造函数。

```
XmlParser* xml_parser_create(void);
```

- 为XmlParser设置builder对象。

```
void xml_parser_set_builder(XmlParser* thiz, XmlBuilder* builder);
```

- 解析XML。

```
void xml_parser_parse(XmlParser* thiz, const char* xml);
```

- 析构函数。

```
void xml_parser_destroy(XmlParser* thiz);
```

在解析时，解析到相应的标签，就调用XmlBuilder相应的函数。

**解析到起始标签时调用xml_builder_on_start_element**

```
static void xml_parser_parse_start_tag(XmlParser* thiz)
```

```
{
    enum _State
    {
        STAT_NAME,
        STAT_ATTR,
        STAT_END,
    }state = STAT_NAME;

    char* tag_name = NULL;
    const char* start = thiz->read_ptr - 1;

    for(; *thiz->read_ptr != '\0'; thiz->read_ptr++)
    {
        char c = *thiz->read_ptr;

        switch(state)
        {
            case STAT_NAME:

            {
                if(isspace(c) || c == '>' || c == '/')
                {
                    tag_name = (char*)xml_parser_strdup(thiz, start,
                                thiz->read_ptr - start);
                    state = (c != '>' && c != '/') ? STAT_ATTR : STAT_END;
                }
                break;
            }
            case STAT_ATTR:
            {
                xml_parser_parse_attrs(thiz, '/');
                state = STAT_END;

                break;
            }
            default:break;
        }

        if(state == STAT_END)
        {
            break;
        }
    }

    tag_name = thiz->buffer + (size_t)tag_name;
    /*解析完成，调用builder的函数xml_builder_on_start_element。*/
    xml_builder_on_start_element(thiz->builder, tag_name, (const char**)
    thiz->attrs);

    if(thiz->read_ptr[0] == '/')
    {
```

```
        /*如果标签以“/”结束，调用builder的函数xml_builder_on_end_element。*/
        xml_builder_on_end_element(thiz->builder, tag_name);
    }

    for(; *thiz->read_ptr != '>' && *thiz->read_ptr != '\0'; thiz->read_ptr++);

    return;
}
```

**解析到结束标签时调用xml_builder_on_end_element**

```
static void xml_parser_parse_end_tag(XmlParser* thiz)
{
    char* tag_name = NULL;
    const char* start = thiz->read_ptr;
    for(; *thiz->read_ptr != '\0'; thiz->read_ptr++)
    {
        if(*thiz->read_ptr == '>')
        {
            tag_name = thiz->buffer + xml_parser_strdup(thiz, start,
                       thiz->read_ptr-start);
            /*解析完成，调用builder的函数xml_builder_on_end_element。*/
            xml_builder_on_end_element(thiz->builder, tag_name);

            break;
        }
    }

    return;
}
```

**解析到文本时调用xml_builder_on_text**

```
static void xml_parser_parse_text(XmlParser* thiz)
{
    const char* start = thiz->read_ptr - 1;
    for(; *thiz->read_ptr != '\0'; thiz->read_ptr++)
    {
        char c = *thiz->read_ptr;

        if(c == '<')
        {
            if(thiz->read_ptr > start)
            {
                /*解析完成，调用builder的函数xml_builder_on_text。*/
                xml_builder_on_text(thiz->builder, start, thiz->read_ptr-start);
            }
            thiz->read_ptr--;
            return;
        }
        else if(c == '&')
```

```
        {
            xml_parser_parse_entity(thiz);
        }
    }

    return;
}
```

**解析到注释时调用xml_builder_on_comment**

```
static void xml_parser_parse_comment(XmlParser* thiz)
{
    enum _State
    {
        STAT_COMMENT,
        STAT_MINUS1,
        STAT_MINUS2,
    }state = STAT_COMMENT;

    const char* start = ++thiz->read_ptr;
    for(; *thiz->read_ptr != '\0'; thiz->read_ptr++)
    {
        char c = *thiz->read_ptr;

        switch(state)
        {
            case STAT_COMMENT:
            {
                if(c == '-')
                {
                    state = STAT_MINUS1;
                }
                break;
            }
            case STAT_MINUS1:
            {
                if(c == '-')
                {
                    state = STAT_MINUS2;
                }
                else
                {
                    state = STAT_COMMENT;
                }
                break;
            }
            case STAT_MINUS2:
            {
                if(c == '>')
                {
                    /*解析完成，调用builder的函数xml_builder_on_comment。*/
                    xml_builder_on_comment(thiz->builder, start,
                    thiz->read_ptr-start-2);
```

```
                return;
            }
        }
        default:break;
    }
}

return;
}
```

**解析到处理指令时调用xml_builder_on_pi_element**

```
static void xml_parser_parse_pi(XmlParser* thiz)
{
    enum _State
    {
        STAT_NAME,
        STAT_ATTR,
        STAT_END
    }state = STAT_NAME;

    char* tag_name = NULL;
    const char* start = thiz->read_ptr;

    for(; *thiz->read_ptr != '\0'; thiz->read_ptr++)
    {
        char c = *thiz->read_ptr;

        switch(state)
        {
            case STAT_NAME:
            {
                if(isspace(c) || c == '>')
                {
                    tag_name = (char*)xml_parser_strdup(thiz, start,
                                thiz->read_ptr - start);
                    state = c != '>' ? STAT_ATTR : STAT_END;
                }
                break;
            }
            case STAT_ATTR:
            {
                xml_parser_parse_attrs(thiz, '?');
                state = STAT_END;
                break;
            }
            default:break;
        }

        if(state == STAT_END)
        {
            break;
        }
```

```
    }

    tag_name = thiz->buffer + (size_t)tag_name;
    /*解析完成，调用builder的函数xml_builder_on_pi_element。*/
    xml_builder_on_pi_element(thiz->builder, tag_name, (const char**)thiz->attrs);

    for(; *thiz->read_ptr != '>' && *thiz->read_ptr != '\0'; thiz->read_ptr++);

    return;
    }
```

从上面的代码可以看出，XmlParser在适当的时候调用了XmlBuilder的接口函数，至于XmlBuilder在这些函数里做什么，要看具体的Builder实现了。

先看一个最简单的XmlBuilder实现，它只是在屏幕上打印出传递给它的数据。

**创建函数**

```
XmlBuilder* xml_builder_dump_create(FILE* fp)
{
    XmlBuilder* thiz = (XmlBuilder*)calloc(1, sizeof(XmlBuilder));

    if(thiz != NULL)
    {
        PrivInfo* priv = (PrivInfo*)thiz->priv;

        thiz->on_start_element   = xml_builder_dump_on_start_element;
        thiz->on_end_element     = xml_builder_dump_on_end_element;
        thiz->on_text            = xml_builder_dump_on_text;
        thiz->on_comment         = xml_builder_dump_on_comment;
        thiz->on_pi_element      = xml_builder_dump_on_pi_element;
        thiz->on_error           = xml_builder_dump_on_error;
        thiz->destroy            = xml_builder_dump_destroy;

        priv->fp = fp != NULL ? fp : stdout;
    }

    return thiz;
}
```

和其他接口的创建函数一样，它只是把接口要求的函数指针指到具体的实现函数上。

**实现on_start_element**

```
static void xml_builder_dump_on_start_element(XmlBuilder* thiz, const char* tag,
const char** attrs)
{
    int i = 0;
    PrivInfo* priv = (PrivInfo*)thiz->priv;
    fprintf(priv->fp, "<%s", tag);

    for(i = 0; attrs != NULL && attrs[i] != NULL && attrs[i + 1] != NULL; i += 2)
```

```
    {
        fprintf(priv->fp, " %s=\"%s\"", attrs[i], attrs[i + 1]);
    }
    fprintf(priv->fp, ">");

    return;
}
```

### 实现on_end_element

```
static void xml_builder_dump_on_end_element(XmlBuilder* thiz, const char* tag)
{
    PrivInfo* priv = (PrivInfo*)thiz->priv;
    fprintf(priv->fp, "\n", tag);

    return;
}
```

### 实现on_text

```
static void xml_builder_dump_on_text(XmlBuilder* thiz, const char* text, size_t length)
{
    PrivInfo* priv = (PrivInfo*)thiz->priv;
    fwrite(text, length, 1, priv->fp);

    return;
}
```

### 实现on_comment

```
static void xml_builder_dump_on_comment(XmlBuilder* thiz, const char* text,
size_t length)
{
    PrivInfo* priv = (PrivInfo*)thiz->priv;
    fprintf(priv->fp, "\n");

    return;
}
```

### 实现on_pi_element

```
static void xml_builder_dump_on_pi_element(XmlBuilder* thiz, const char* tag,
const char** attrs)
{
    int i = 0;
    PrivInfo* priv = (PrivInfo*)thiz->priv;
    fprintf(priv->fp, "fp, " %s=\"%s\"", attrs[i], attrs[i + 1]);

    fprintf(priv->fp, "?>\n");

    return;
}
```

**实现on_error**

```
static void xml_builder_dump_on_error(XmlBuilder* thiz, int line, int row,
const char* message)
{
    fprintf(stderr, "(%d,%d) %s\n", line, row, message);

    return;
}
```

上面的XmlBuilder实现简单，而且有一定的实用价值，我一般都会先写这样一个Builder。它不但对调试程序有不小的帮助，而且只要稍做修改，就可以把它改进成一个美化数据格式的小工具，不管原始数据的格式（当然要合符相应的语法规则）有多乱，你都能以一种比较好看的方式打印出来。

下面我们再看一个比较复杂的XmlBuilder的实现，它根据接收的数据构建一棵XML树。

**创建函数**

```
XmlBuilder* xml_builder_tree_create(void)
{
    XmlBuilder* thiz = (XmlBuilder*)calloc(1, sizeof(XmlBuilder));

    if(thiz != NULL)
    {
        PrivInfo* priv = (PrivInfo*)thiz->priv;

        thiz->on_start_element   = xml_builder_tree_on_start_element;
        thiz->on_end_element     = xml_builder_tree_on_end_element;
        thiz->on_text            = xml_builder_tree_on_text;
        thiz->on_comment         = xml_builder_tree_on_comment;
        thiz->on_pi_element      = xml_builder_tree_on_pi_element;
        thiz->on_error           = xml_builder_tree_on_error;
        thiz->destroy            = xml_builder_tree_destroy;

        priv->root = xml_node_create_normal("__root__", NULL);
        priv->current = priv->root;
    }

    return thiz;
}
```

和其他接口的创建函数一样，它只是把接口要求的函数指针指到具体的实现函数上。这里还创建了一个根结点__root__，以保证整棵树只有一个根结点。

**实现on_start_element**

```
static void xml_builder_tree_on_start_element(XmlBuilder* thiz, const char* tag,
const char** attrs)
{
```

```
    XmlNode* new_node = NULL;
    PrivInfo* priv = (PrivInfo*)thiz->priv;

    new_node = xml_node_create_normal(tag, attrs);
    xml_node_append_child(priv->current, new_node);
    priv->current = new_node;

    return;
}
```

这里创建了一个新的结点，并追加为priv->current的子结点，然后让priv->current指向新的结点。

**实现on_end_element**

```
static void xml_builder_tree_on_end_element(XmlBuilder* thiz, const char* tag)
{
    PrivInfo* priv = (PrivInfo*)thiz->priv;
    priv->current = priv->current->parent;
    assert(priv->current != NULL);

    return;
}
```

这里只是让priv->current指向它的父结点。

**实现on_text**

```
static void xml_builder_tree_on_text(XmlBuilder* thiz, const char* text, size_t length)
{
    XmlNode* new_node = NULL;
    PrivInfo* priv = (PrivInfo*)thiz->priv;

    new_node = xml_node_create_text(text);
    xml_node_append_child(priv->current, new_node);

    return;
}
```

这里创建一个文本结点，并追加为priv->current的子结点。

**实现on_comment**

```
static void xml_builder_tree_on_comment(XmlBuilder* thiz, const char* text,
    size_t length)
{
    XmlNode* new_node = NULL;
    PrivInfo* priv = (PrivInfo*)thiz->priv;

    new_node = xml_node_create_comment(text);
```

```
    xml_node_append_child(priv->current, new_node);

    return;
}
```

这里创建一个注释结点，并追加为priv->current的子结点。

### 实现on_pi_element

```
static void xml_builder_tree_on_pi_element(XmlBuilder* thiz, const char* tag,
const char** attrs)
{
    XmlNode* new_node = NULL;
    PrivInfo* priv = (PrivInfo*)thiz->priv;

    new_node = xml_node_create_pi(tag, attrs);
    xml_node_append_child(priv->current, new_node);

    return;
}
```

这里创建一个处理指令结点，并追加为priv->current的子结点。

### 实现on_error

```
static void xml_builder_tree_on_error(XmlBuilder* thiz, int line, int row,
const char* message)
{
    fprintf(stderr, "(%d,%d) %s\n", line, row, message);

return;
}
```

下面我们再看XmlNode的数据结构和主要函数。

### 数据结构

```
typedef struct _XmlNode
{
        XmlNodeType type;
        union
        {
            char* text;
            char* comment;
            XmlNodePi pi;
            XmlNodeNormal normal;
        }u;
        struct _XmlNode* parent;
        struct _XmlNode* children;
        struct _XmlNode* sibling;
}XmlNode;
```

type决定了结点的类型，可以是处理指令（XML_NODE_PI）、文本（XML_NODE_TEXT）、注释（XML_NODE_COMMENT）或普通标签（XML_NODE_NORMAL）。

联合体用于存放具体结点信息。

parent指向父结点。

children指向第一个子结点。

sibling指向下一个兄弟结点。

**创建普通标签结点**

```
XmlNode* xml_node_create_normal(const char* name, const char** attrs)
{
        XmlNode* node = NULL;
        return_val_if_fail(name != NULL, NULL);

        if((node = calloc(1, sizeof(XmlNode))) != NULL)
        {
            int i = 0;
            node->type = XML_NODE_NORMAL;
            node->u.normal.name = strdup(name);

            if(attrs != NULL)
            {
                    for(i = 0; attrs[i] != NULL && attrs[i+1] != NULL; i += 2)
                    {
                xml_node_append_attr(node, attrs[i], attrs[i+1]);
                    }
            }
        }

        return node;
}
```

**创建处理指令结点**

```
XmlNode* xml_node_create_pi(const char* name, const char** attrs)
{
        XmlNode* node = NULL;
        return_val_if_fail(name != NULL, NULL);

        if((node = calloc(1, sizeof(XmlNode))) != NULL)
        {
            int i = 0;
            node->type = XML_NODE_PI;
            node->u.pi.name = strdup(name);
            if(attrs != NULL)
            {
                for(i = 0; attrs[i] != NULL && attrs[i+1] != NULL; i += 2)
                {
                xml_node_append_attr(node, attrs[i], attrs[i+1]);
```

```
                }
            }
        }

        return node;
}
```

### 创建文本结点

```
XmlNode* xml_node_create_text(const char* text)
{
        XmlNode* node = NULL;
        return_val_if_fail(text != NULL, NULL);

        if((node = calloc(1, sizeof(XmlNode))) != NULL)
        {
            node->type = XML_NODE_TEXT;
            node->u.text = strdup(text);
        }

        return node;
}
```

### 创建注释结点

```
XmlNode* xml_node_create_comment(const char* comment)
{
        XmlNode* node = NULL;
        return_val_if_fail(comment != NULL, NULL);

        if((node = calloc(1, sizeof(XmlNode))) != NULL)
        {
            node->type = XML_NODE_COMMENT;
            node->u.comment = strdup(comment);
        }

        return node;
}
```

### 追加一个兄弟结点

```
XmlNode* xml_node_append_sibling(XmlNode* node, XmlNode* sibling)
{
        return_val_if_fail(node != NULL && sibling != NULL, NULL);

        if(node->sibling == NULL)
        {
            /*没有兄弟结点，让兄弟结点指向sibling */
            node->sibling = sibling;
        }
        else
        {
```

```
        /*否则，把sibling追加为最后一个兄弟结点*/
        XmlNode* iter = node->sibling;
        while(iter->sibling != NULL) iter = iter->sibling;
        iter->sibling = sibling;
    }
    /*让兄弟结点的父结点指向自己的父结点*/

    sibling->parent = node->parent;

    return sibling;
}
```

### 追加一个子结点

```
XmlNode* xml_node_append_child(XmlNode* node, XmlNode* child)
{
    return_val_if_fail(node != NULL && child != NULL, NULL);

    if(node->children == NULL)
    {
        /*没有子结点，让子结点指向child */
        node->children = child;
    }
    else
    {
        /*否则，把child 追加为最后一个子结点*/
        XmlNode* iter = node->children;
        while(iter->sibling != NULL) iter = iter->sibling;
        iter->sibling = child;
    }
    /*让子结点的父结点指向自己*/

    child->parent = node;

    return child;
}
```

回头再看一下XmlParser、XmlBuilder及几个具体的XmlBuilder的实现，我们可以看到，它们的实现都非常简单，其实这完全得益于Builder模式的设计方法。它利用分而治之的思想，把数据的解析和数据的处理分开，降低了实现的复杂度。其次它利用了抽象的思想，从而使数据的解析只关心完成数据处理的接口，而不关心的它的实现，使得数据解析和数据处理可以独立变化。

分而治之和抽象是降低复杂度最有效的手段之一，它们在Builder模式里得到了很好的体现。初学者应该多花些时间去体会。



## 10.3 管道过滤器模式

按照POSA（《面向模式的软件架构》）里的说法，管道过滤器（Pipe-And-Filter）应该属于

架构模式，因为它通常决定了一个系统的基本架构。管道过滤器和生产流水线类似，在生产流水线上，原材料在流水线上经一道一道的工序，最后形成某种有用的产品。而在管道过滤器中，数据经过一个一个的过滤器，最后得到需要的数据。

**基本的管道过滤器**

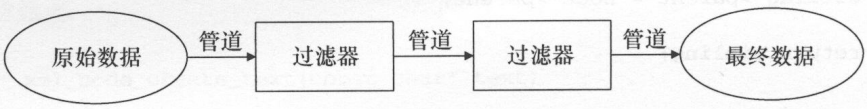



图10-8　基本的管道过滤器

管道负责数据的传递，它把原始数据传递给第一个过滤器，把一个过滤器的输出传递给下一个过滤器作为下一个过滤器的输入，并重复这个过程直到处理结束。要注意的是，管道只是对数据传输的抽象，它可能是管道，也可能是其他通信方式，甚至可能什么都没有（所有过滤器都在原始数据基础上进行处理）。

过滤器负责数据的处理，过滤器可以有多个，每个过滤器对数据做特定的处理，它们之间没有依赖关系，一个过滤器不必知道其他过滤器的存在。这种松耦合的设计，使得过滤器只需要实现单一的功能，从而降低了系统的复杂度，也使得过滤器之间依赖最小，从而以更加灵活的组合来实现新的功能。

编译器就是基于管道过滤器模式设计的（如图10-9所示）。

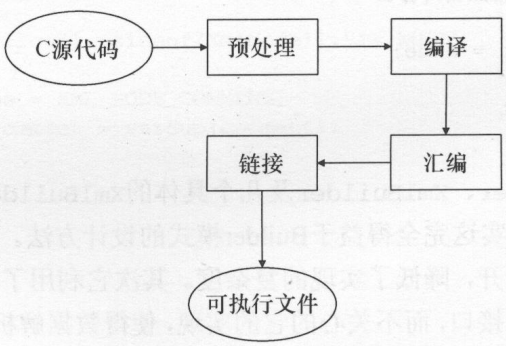



图10-9　基于管道过滤器模式的编译器设计

输入：源程序。

预处理：负责宏展开和去掉注释等工作。

编译：进行词法分析、语法分析、语义分析、代码优化和代码产生。

汇编：负责把汇编代码转换成机器指令，生成目标文件。

链接：负责把多个目标文件、静态库和共享库链接成可执行文件/共享库。

输出：可执行文件/共享库。

### 复合过滤器

过滤器可以由多个其他过滤器组合起来的，比如上面的“编译”过程可以认为是一个复合过滤器，如图10-10所示。

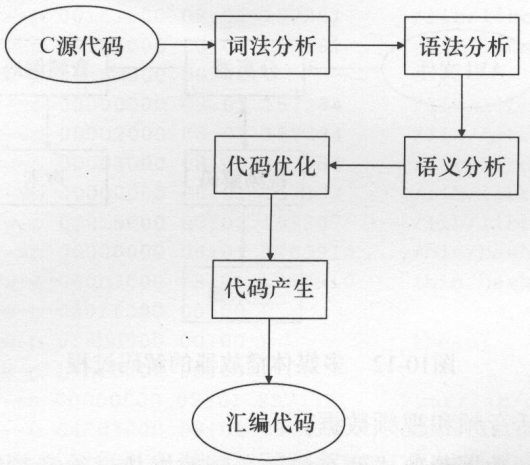



图10-10 “编译”过程的复合过滤器

输入：预处理之后的源代码。
词法分析：负责将源程序分解成一个一个的token，这些token是组成源程序的基本单元。
语法分析：把词法分析得到的token解析成语法树。
语义分析：对语法树进行类型检查等语义分析。
代码优化：对语法树进行重组和修改，以优化代码的速度和大小。
代码产生：根据语法树产生汇编代码。
输出：汇编代码。

### 支持多个输入的过滤器

过滤器可以有多个输入。比如上面“链接”，它能接收多个输入，如图10-11所示。

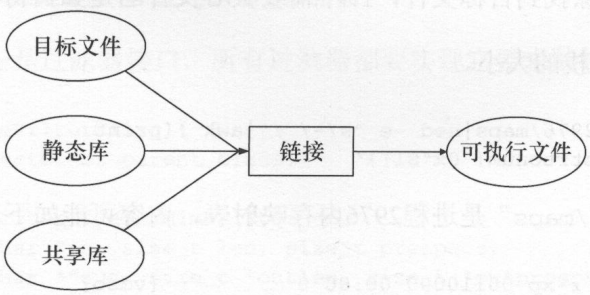



图10-11 支持多个输入的“链接”过滤器

“链接”过滤器能接收多个数据源，如目标文件、静态库和共享库。

**具有多个输出的过滤器**

过滤器可以有多个输出，例如多媒体播放器的解码过程（如图10-12所示）。

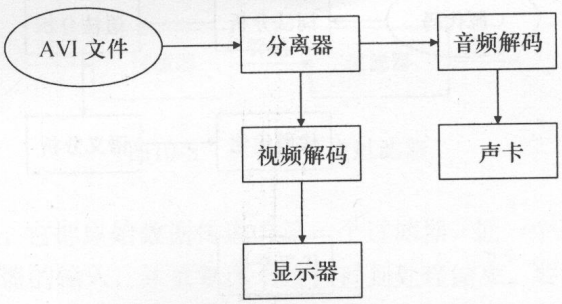



图10-12　多媒体播放器的解码过程

输入：AVI文件，包括音频和视频数据。

分离器：把音频和视频数据分离成两个流，音频数据传递给音频解码器，视频数据传递给视频解码器。

音频解码器：把压缩的音频数据解码成原始的音频数据。

视频解码器：把压缩的视频数据解码成原始的图像数据。

输出：音频数据传递给声卡，图像数据传递给显示器。

管道过滤器是最贴近程序员生活的模式，也是类Unix系统的基本设计理念之一。作为Linux下的程序员，我们天天都使用这个模式。比如

(1) 删除当前目录及子目录下的目标文件。

```
find -name \*.o|xargs rm -f
```

find是过滤器：它找出所有目标文件，它不需要关心查找文件的目的。

rm是过滤器：它删除找到目标文件，rm不需要关心文件名是如何得来的。

(2) 查看某个进程的栈的大小。

```
grep stack /proc/2976/maps|sed -e "s/-/ /"|awk '{print
strtonum("0x"$2)-strtonum("0x"$1)}'
```

其中“/proc/2976/maps”是进程2976内存映射表，内容可能如下。

```
00110000-00111000 r-xp 00110000 00:00 0          [vdso]
00111000-0011b000 r-xp 00000000 08:01 154857     /lib/libnss_files-2.8.so
```

```
0011b000-0011c000 r--p 0000a000 08:01 154857     /lib/libnss_files-2.8.so
0011c000-0011d000 rw-p 0000b000 08:01 154857     /lib/libnss_files-2.8.so
00907000-00923000 r-xp 00000000 08:01 157280     /lib/ld-2.8.so
00923000-00924000 r--p 0001c000 08:01 157280     /lib/ld-2.8.so
00924000-00925000 rw-p 0001d000 08:01 157280     /lib/ld-2.8.so
00927000-00a8a000 r-xp 00000000 08:01 157281     /lib/libc-2.8.so
00a8a000-00a8c000 r--p 00163000 08:01 157281     /lib/libc-2.8.so
00a8c000-00a8d000 rw-p 00165000 08:01 157281     /lib/libc-2.8.so
00a8d000-00a90000 rw-p 00a8d000 00:00 0
00abd000-00ac0000 r-xp 00000000 08:01 157284     /lib/libdl-2.8.so
00ac0000-00ac1000 r--p 00002000 08:01 157284     /lib/libdl-2.8.so
00ac1000-00ac2000 rw-p 00003000 08:01 157284     /lib/libdl-2.8.so
0383f000-03855000 r-xp 00000000 08:01 157307     /lib/libtinfo.so.5.6
03855000-03858000 rw-p 00015000 08:01 157307     /lib/libtinfo.so.5.6
08047000-080fa000 r-xp 00000000 08:01 1180910    /bin/bash
080fa000-080ff000 rw-p 000b3000 08:01 1180910    /bin/bash
080ff000-08104000 rw-p 080ff000 00:00 0
088bd000-088ff000 rw-p 088bd000 00:00 0          [heap]
b7bfb000-b7bfd000 rw-p b7bfb000 00:00 0
b7bfd000-b7c04000 r--s 00000000 08:01 237138     /usr/lib/gconv/gconv-modules.cache
b7c04000-b7d1e000 r--p 047d3000 08:01 237437     /usr/lib/locale/locale-archive
b7d1e000-b7d5e000 r--p 0236e000 08:01 237437     /usr/lib/locale/locale-archive
b7d5e000-b7f5e000 r--p 00000000 08:01 237437     /usr/lib/locale/locale-archive
b7f5e000-b7f60000 rw-p b7f5e000 00:00 0
bfe5e000-bfe73000 rw-p bffeb000 00:00 0          [stack]
```

grep是过滤器：它从文件/proc/2976/maps里找到下面这行数据。

```
bfe5e000-bfe73000 rw-p bffeb000 00:00 0 [stack]
```

sed是过滤器：它把“-”替换成“ ”（长度为一个字符的空格），数据变成下面的内容。

```
bfe5e000 bfe73000 rw-p bffeb000 00:00 0 [stack]
```

awk是过滤器：它会计算0xbfe73000和0x bfe5e000的差值，并打印出来。

下面我们来看看管道过滤器在程序里的实现方式。这里我们以TinyMail为例，TinyMail是一款针对移动设备定制的邮件客户端软件。它使用Camel-lite完成邮件内容解析和传输。Camel-lite对邮件内容的处理大体上是基于管道过滤器模式的。

CamelMimeFilter是过滤器接口，所有过滤器都要实现它要求的接口函数。

```
struct _CamelMimeFilterClass {
        CamelObjectClass parent_class;

        void (*filter)(CamelMimeFilter *f,
               char *in, size_t len, size_t prespace,
               char **out, size_t *outlen, size_t *outprespace);
        void (*complete)(CamelMimeFilter *f,
```

```
                char *in, size_t len, size_t prespace,
                char **out, size_t *outlen, size_t *outprespace);
        void (*reset)(CamelMimeFilter *f);
};
```

CamelMimeFilterClass是从CamelObjectClass继承过来的。这里的接口定义和我们前面所讲的接口定义有些差别，但原理上都是一样的，通过函数指针来抽象具体的功能。这里要求实现以下三个接口函数。

filter：过滤器的处理函数。

complete：过滤器的处理函数。与filter不同的是，调用complete之后不能调其他filter。

reset：重置当前filter的状态。

Camel实现了很多Filter，其中CamelMimeFilterBasic实现了邮件基本的编码和解析功能。它的filter函数实现如下。

```
static void filter(CamelMimeFilter *mf, char *in, size_t len, size_t prespace,
char **out, size_t *outlen, size_t *outprespace)
{
        CamelMimeFilterBasic *f = (CamelMimeFilterBasic *)mf;
        size_t newlen;

        switch(f->type) {
        case CAMEL_MIME_FILTER_BASIC_BASE64_ENC:
            /* wont go to more than 2x size (overly conservative) */
            camel_mime_filter_set_size(mf, len*2+6, FALSE);
            newlen = g_base64_encode_step((const guchar *) in, len, TRUE, mf->outbuf,
                &f->state, &f->save);
            g_assert(newlen <= len*2+6);
            break;
        case CAMEL_MIME_FILTER_BASIC_QP_ENC:
            /* *4 is overly conservative, but will do */
            camel_mime_filter_set_size(mf, len*4+4, FALSE);
            newlen = camel_quoted_encode_step((unsigned char *) in, len,
                (unsigned char *) mf->outbuf, &f->state, (gint *) &f->save);
            g_assert(newlen <= len*4+4);
            break;
        case CAMEL_MIME_FILTER_BASIC_UU_ENC:
            /* won't go to more than 2 * (x + 2) + 62 */
            camel_mime_filter_set_size (mf, (len + 2) * 2 + 62, FALSE);
            newlen = camel_uuencode_step ((unsigned char *) in, len, (unsigned char *)
                 mf->outbuf, f->uubuf, &f->state, (guint32*) &f->save);
            g_assert (newlen <= (len + 2) * 2 + 62);
            break;
        case CAMEL_MIME_FILTER_BASIC_BASE64_DEC:
            /* output can't possibly exceed the input size */
            camel_mime_filter_set_size(mf, len+3, FALSE);
            newlen = g_base64_decode_step(in, len, (guchar *) mf->outbuf, &f->state,
                (guint *) &f->save);
```

```
        g_assert(newlen <= len+3);
        break;
    case CAMEL_MIME_FILTER_BASIC_QP_DEC:
        /* output can't possibly exceed the input size */
        camel_mime_filter_set_size(mf, len + 2, FALSE);
        newlen = camel_quoted_decode_step((unsigned char *) in, len,
            (unsigned char *) mf->outbuf, &f->state, (gint *) &f->save);
        g_assert(newlen <= len + 2);
        break;
    case CAMEL_MIME_FILTER_BASIC_UU_DEC:
        if (!(f->state & CAMEL_UUDECODE_STATE_BEGIN)) {
            register char *inptr, *inend;
            size_t left;

            inptr = in;
            inend = inptr + len;

            while (inptr < inend) {
            left = inend - inptr;
            if (left < 6) {
                if (!strncmp (inptr, "begin ", left))
                    camel_mime_filter_backup (mf, inptr, left);
                break;
            } else if (!strncmp (inptr, "begin ", 6)) {
                for (in = inptr; inptr < inend && *inptr != '\n'; inptr++);
                if (inptr < inend) {
                    inptr++;
                    f->state |= CAMEL_UUDECODE_STATE_BEGIN;
                    /* we can start uudecoding... */
                    in = inptr;
                    len = inend - in;
                } else {
                    camel_mime_filter_backup (mf, in, left);
                }
                break;

            }

            /* go to the next line */
            for ( ; inptr < inend && *inptr != '\n'; inptr++);

            if (inptr < inend)
                inptr++;
            }
        }

        if ((f->state & CAMEL_UUDECODE_STATE_BEGIN) && !(f->state &
        CAMEL_UUDECODE_STATE_END)) {
            /* "begin <mode> <filename>\n" has been found,
            so we can now start decoding */
            camel_mime_filter_set_size (mf, len + 3, FALSE);
            newlen = camel_uudecode_step ((unsigned char *) in, len,
```

```
                    (unsigned char *) mf->outbuf, &f->state, (guint32 *) &f->save);
            } else {
                newlen = 0;
            }
            break;
        default:
            g_warning ("unknown type %u in CamelMimeFilterBasic", f->type);
            goto donothing;
        }

        *out = mf->outbuf;
        *outlen = newlen;
        *outprespace = mf->outpre;

        return;
donothing:
        *out = in;
        *outlen = len;
        *outprespace = prespace;
}
```

这个过滤器实现了下面三种编码方式的编码和解码：

(1) UU（Unix-to-Unix encoding）；

(2) Base64；

(3) QP（Quote-Printable）。

Camel还提供了其他一些过滤器，如

CamelMimeFilterGZip：压缩和解压。

CamelMimeFilterHTML：去掉HTML标签。

CamelMimeFilterCRLF：使用\r\n作为换行符。

CamelMimeFilterToHTML：加上HTML标签。

CamelMimeFilterCharset：字符集转换。

所有的过滤器由CamelStreamFilter来组合，CamelStreamFilter提供了下面两个函数。

(1) 增加过滤器。

```
int camel_stream_filter_add (CamelStreamFilter *stream, CamelMimeFilter *filter);
```

(2) 移除过滤器。

```
void camel_stream_filter_remove (CamelStreamFilter *stream, int id);
```

CamelStreamFilter实现了CamelStream接口，这里应用了前面所讲的装饰模式，它不改变CamelStream的接口，但给CamelStream加上了数据转换功能，CamelStreamFilter的创建函

数如下。

```
CamelStreamFilter *camel_stream_filter_new_with_stream (CamelStream *stream);
```

传入一个CamelStream对象，然后对这个对象进行装饰。在读写数据时，调用相应的Filter，下面是写函数的实现。

```
do_read (CamelStream *stream, char *buffer, size_t n)
{
        CamelStreamFilter *filter = (CamelStreamFilter *)stream;
        struct _CamelStreamFilterPrivate *p = _PRIVATE(filter);
        ssize_t size;
        struct _filter *f;

        p->last_was_read = TRUE;

        g_check(p->realbuffer);

        if (p->filteredlen<=0) {
            size_t presize = READ_PAD;

            size = camel_stream_read(filter->source, p->buffer, READ_SIZE);
            if (size <= 0) {
                /* this is somewhat untested */
                if (camel_stream_eos(filter->source)) {
                f = p->filters;
                p->filtered = p->buffer;
                p->filteredlen = 0;
                while (f) {
                    camel_mime_filter_complete(f->filter, p->filtered,
                    p->filteredlen, presize, &p->filtered, &p->filteredlen,
                    &presize);
                    g_check(p->realbuffer);
                    f = f->next;
                }
                size = p->filteredlen;
                p->flushed = TRUE;
                }
                if (size <= 0)
                return size;
            } else {
                f = p->filters;
                p->filtered = p->buffer;
                p->filteredlen = size;
                d(printf ("\n\nOriginal content (%s): '", ((CamelObject *)filter->source)
            ->klass->name));
                d(fwrite(p->filtered, sizeof(char), p->filteredlen, stdout));
                d(printf("'\n"));

                while (f) {
                camel_mime_filter_filter(f->filter, p->filtered, p->filteredlen,
                presize, &p->filtered, &p->filteredlen, &presize);
```

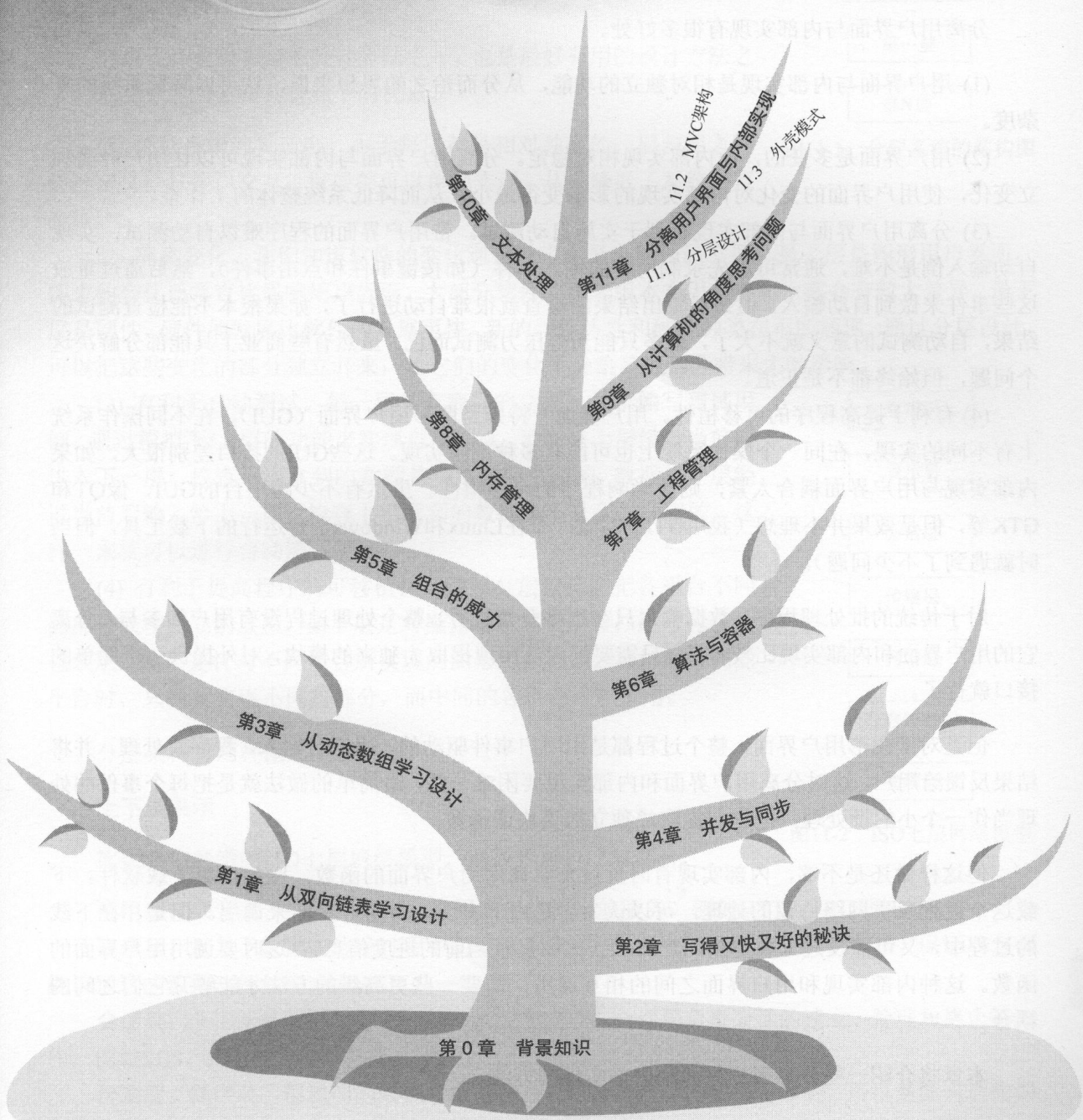
```
                g_check(p->realbuffer);

                d(printf ("Filtered content (%s): '", ((CamelObject *)f->filter)
                ->klass->name));
                d(fwrite(p->filtered, sizeof(char), p->filteredlen, stdout));
                d(printf("'\n"));

                f = f->next;
                }
            }
        }

        size = MIN(n, p->filteredlen);
        memcpy(buffer, p->filtered, size);
        p->filteredlen -= size;
        p->filtered += size;

        g_check(p->realbuffer);

        return size;
}
```

这里先调用camel_stream_read读取数据，然后依次调用fitler对数据进行处理，最后把数据返回给调用者。do_write的过程类似，CamelStreamFilter对编码和解码都支持，而使用者不用关心。

管道过滤器模式应用相当广泛，它不仅能进行文本数据处理，任何以数据处理为中心的系统都可以用管道过滤器模式作为基本架构。

# 第 11 章

# 分离用户界面与内部实现



用户界面就是与用户交互的接口，通常包括输入和输出（显示）两个部分。用户使用键盘等输入设备把数据输入给程序，程序做相应处理后，再将结果输出到显示器或其他设备上。所谓的内部实现（也称为内部逻辑）就是负责数据处理的这一部分功能，它占的比例最大且实现也最复杂。

分离用户界面与内部实现有很多好处。

(1) 用户界面与内部实现是相对独立的功能，从分而治之的思想来说，这可以降低系统的复杂度。

(2) 用户界面是多变的，而内部实现相对稳定。分离用户界面与内部实现可以让用户界面独立变化，使用户界面的变化对内部实现的影响变得最小，从而降低系统整体的工作量。

(3) 分离用户界面与内部实现有利于实施自动测试。带用户界面的程序难以自动测试，实现自动输入倒是不难，通常可以先录制所有的输入事件（如按键事件和点击事件），然后通过重放这些事件来做到自动输入。但是对输出结果的检查就很难自动进行了，如果根本不能检查测试的结果，自动测试的意义就不大了，最多只能当作压力测试而已。虽然有些商业工具能部分解决这个问题，但始终都不是正道。

(4) 有利于提高程序的可移植性。用户界面，特别是图形用户界面（GUI），在不同操作系统上有不同的实现，在同一个操作系统上也可能有多种不同实现。这些GUI的接口差别很大，如果内部实现与用户界面耦合太紧，则会影响程序的可移植性。虽然有不少跨平台的GUI，像QT和GTK等，但是效果并不理想（我用GTK+写过一个在Linux和Windows平台运行的下载工具，但当时就遇到了不少问题）。

对于传统的批处理程序，数据输入只在程序开始进行，整个处理过程没有用户的参与，分离它的用户界面和内部实现比较容易，只需要把内部实现提取为独立的模块，对外提供一个简单的接口就行了。

但是对于图形用户界面，整个过程都是由用户事件驱动的，用户有输入，程序就处理，并将结果反馈给用户。这时分离用户界面和内部实现要困难一些，最简单的做法就是把每个事件的处理当作一个小的批处理，把它们提取成独立的模块或函数。

但这样做还是不够，内部实现有时反过来要调用用户界面的函数。比如一个下载软件，下载这个过程负责网络协议的处理，可以认为它是内部实现，由用户界面来调用。但是，在下载的过程中，又可能反过来要更新用户界面（比如显示当前的进度信息），这时要调用用户界面的函数。这种内部实现和用户界面之间的相互调用，需要一些更高级的方法才能解开它们之间的耦合。

本章将介绍一些分离用户界面和内部实现的方法。

# 11.1 分层设计

分层设计也是分而治之的思想的应用。它把整体分成多个部分，但各部分之间的地位并不是等同的，而是某种上下级的关系。分层系统的结构如图11-1所示。

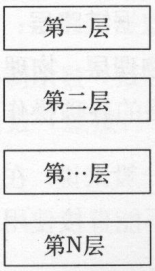



图11-1 分层系统的结构图

分层设计是最古老的设计方法之一，也是最好有用的设计方法之一。基于分层的架构具有如下的优点。

(1) 降低系统的复杂度。由于每层都是相对独立的，层与层之间通过定义良好接口交互，每层都可以单独实现，从而降低了系统的复杂度。

(2) 隔离变化。我们知道软件的变化通常发生在最上层和最下层。最上层是图形用户界面，需求的变化通常直接影响用户界面，大部分软件的新老版本在用户界面上都会有较大差异。最下层是硬件，硬件的发展比软件的发展更快，新的硬件特性和硬件种类在不断涌现。通过分层设计，可以把这些变化的部分独立开来，让它们的变化不会给其他部分带来大的影响。

(3) 有利于自动测试。每一层具有独立的功能，易于编写测试用例。通过模拟下/上层的功能来实现自动测试：数据从上层经过当前层进入下一层，检查下层拿到的数据是不是我们期望的；数据从下层经过当前层输入给上一层，检查上层拿到的数据是不是我们期望的，这样一来就可以进行自动测试了。

(4) 有利于提高程序的可移植性。通过分层设计，把各平台不同的部分放在独立的层里，如下层是对操作系统提供的接口进行包装的包装层，上层是针对不同平台所实现的图形用户界面。移植到不同的平台时，只需要实现不同的部分，而中间的各层都可以重用。

分层架构有两种图形表示方法。

**上下层表示**

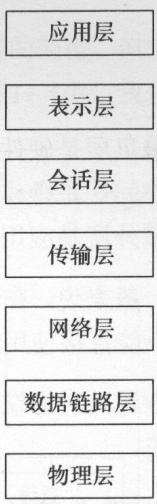



图11-2 ISO七层网络模型

比如我们熟悉的ISO七层网络模型，可以用图11-2来表示。

**应用层**：通常是一些应用程序，为用户提供特定的服务，例如HTTP、FTP和Telnet等。

**表示层**：定义了数据在传输过程中的表示方法，是发送方与接收方之间的契约，保证双方都能识别这些数据，例如XDR和ASN.1。

**会话层**：会话就是网络实体之间的一次完整交互。会话层负责会话的建立、管理和终止等工作，例如TLS、SSH和BSD套接字。

**传输层**：提供对上层透明的端到端数据传输。可以是可靠的或不可靠的，可以是面向连接的

或非面向连接的，例如TCP和UDP。

**网络层**：负责把数据从一个网络实体传递给目的网络实体，例如IP和ICMP。

**数据链路层**：负责把数据从一个网络实体传递给下一个网络实体，例如Ethernet和PPP。

**物理层**：物理层定义了通讯网络之间物理链路的电气或机械特性，以及激活、维护和关闭这条链路的各项操作，例如wire和radio。

一般来说，在这种上下层表示方法中，上层依赖于下层，上层可以使用下层提供的功能，而下层不能直接使用上层提供的功能。

**内外层表示**

比如我们在学习计算机组成原理时，看到的计算机的组成结构是如图11-3所示的。

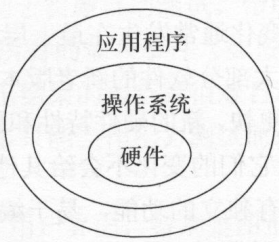



图11-3　计算机的组成结构

最里层是硬件，像主板、显卡、声卡和硬盘等。

中间层是操作系统，它提供对硬件的抽象、进程管理、内存管理和文件管理等功能。

最外层是应用程序，为用户提供具体的功能，如办公软件和浏览器等。

一般来说，在这种内外层表示方法中，外层依赖于内层，外层可以使用内层提供的功能，而内层不能直接使用外层提供的功能。

在这两种方法中，上下层的表示方法更常见，也更容易绘制，后面我们一律采用上下层的表示方法。

## 层与层之间的通信

为了降低层与层之间的耦合，层与层之间的通信必须按一定的规则进行。

(1) 上层可以直接调用下层提供的函数。

(2) 上层可以直接调用相邻下层或间接下层提供的函数，但最好只调用相邻下层所提供的函数。

(3) 下层不能直接调用上层提供的函数。

由于下层不能直接调用上层提供的函数，那下层需要传递数据给上层，就得使用其他途径，通常的做法有两种。

(1) 通过消息传递。消息是抽象的，接收消息的实体（进程或窗口句柄）也是抽象的。下层通过消息把数据传递给上层，它并不直接依赖于上层。但是消息通常是平台相关的，在编写跨平台软件时，这种做法并不可取。

(2) 通过回调函数（或接口）传递。在初始化时，上层给下层提供一个回调函数（或接口），下层在需要主动与上层通信时，就调用这个回调函数（或接口），由于回调函数（或接口）是抽象且平台无关的，因此这种做法是最好的。

下面我们再来讨论用分层架构来分离用户界面与内部实现。

带用户界面的应用程序，我们通常把它分为两层或三层。

如图11-4所示的是两层架构。

**用户界面**：负责接收用户的输入，然后调用内部实现的函数进行处理，最后把处理结果显示给用户。

**内部实现**：负责具体的功能实现，接收来自用户界面的数据，做相应处理，最后把处理结果返回给用户界面。

用户界面处于上层，内部实现处于下层。用户界面可以直接调用内部实现的函数，但内部实现不能反过来直接调用用户界面的函数，否则就违背了分层设计的原则。在图11-4中，我们用实线箭头来表示直接调用，用虚线箭头表示回调或消息。

要注意的是，两层并不代表两个模块。通常每层都会进一步细分，拆分成多个小的模块，甚至某层的内部本身又一个多层架构，如图11-5所示。

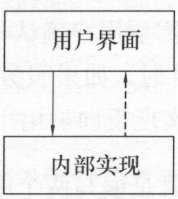



图11-4 两层架构

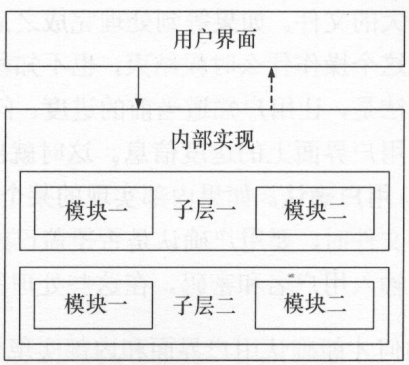



图11-5 一种细分过的两层架构

具体怎么去拆分，要根据实际情况而定。一个简单的电子书阅读软件，可能按下列方式进行拆分，如图11-6所示。

实际的电子书软件可能要复杂得多，但它们的设计方法都是类似的。

如图11-7所示的是三层架构。

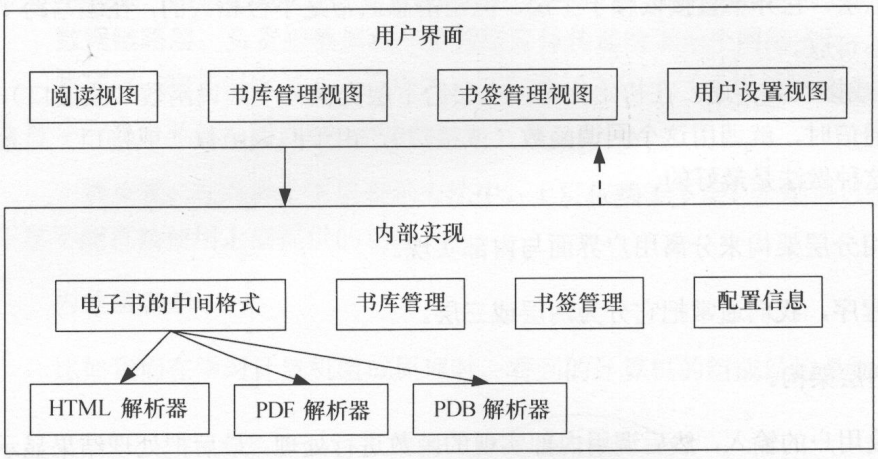



图11-6　两层架构的电子书阅读软件

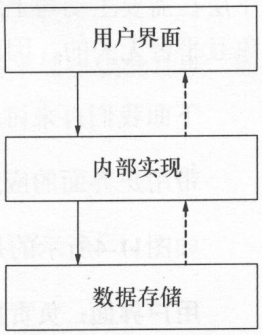



图11-7　三层架构

与两层架构不同的是，这里增加了数据存储层。数据存储的方式有很多种，比如用普通文件、XML文件和数据库管理系统等方式来存储。各个数据库管理系统也或多或少有些差异，把这些不同储存方式放在独立的一层里，让它们的变化不会对上层产生影响。

如果内部实现的模块划分合理，用户界面接收用户的输入，然后调用内部实现的函数进行处理，最后把结果返回给用户。一般来说，在整个过程中不会出现内部实现反过来调用用户界面的情况。但在下列情况下，内部实现反过来调用用户界面是不可避免的。

(1) **长时间的操作**。如果内部实现的某个处理需要比较长的时间才能完成，比如从网络上下载一个大的文件。如果等到处理完成之后，才返回给用户一个结果，那是不太友好的，因为用户不知道这个操作什么时候结束，也不知程序是死掉了还是在运行，他可能因此而感到焦虑。比较好的做法是，让用户知道当前的进度。问题是只有内部实现才知道当前的处理进度，但它不能直接更新用户界面上的进度信息。这时就出现了内部实现反向调用用户界面的情况。

(2) **用户确认**。如果内部实现的某个处理，在处理的过程中，需要用户确认一些操作。比如，在复制文件时，要用户确认是否覆盖已经存在的文件。在发送邮件时，如果服务器要求认证，需要用户输入用户名和密码。在这些处理的过程中，都会出现内部实现反向调用用户界面的情况。

如何才能确认用户界面和内部实现分开了呢？最简单的办法就是编写两个用户界面，一个是图形用户界面，一个是基于终端（命令行）的用户界面，如果在这两种实现中，没有重复的代码，那就说明用户界面和内部实现是真正分开了。

下面我们看一个例子。

Linux的PAM（Pluggable Authentication Modules）是Linux下的一套鉴权（Authentication）机

制。在通常的情况下，鉴权就是对用户名和密码进行验证，以确认用户的身份。当然这是最简单的情况，PAM还有不少其他复杂的特性，由于这不是我们的重点，就不多说了。

在鉴权的时候，不同的鉴权机制所要求的输入信息不一样，这要在鉴权的过程中才能知道。鉴权属于内部实现，它不能直接调用用户界面的功能。拿输入用户名和密码来说，有的是在终端下输入，有的是在图形用户界面下输入，而图形用户界面本身也有多种选择，把这些用户界面的功能耦合到鉴权机制内部是不合适的，这不符合隔离变化的原则。

为了解决这个问题，PAM要求用户界面传入一个接口，通过这个接口来反向调用用户界面，以达到与用户交互的目的。

```
/* The actual conversation structure itself */
struct pam_conv {
        int (*conv)(int num_msg, const struct pam_message **msg,
            struct pam_response **resp, void *appdata_ptr);
        void *appdata_ptr;
};
(libpam/include/security/_pam_types.h)
```

这个接口只有一个接口函数conv，它负责提示用户和接收用户的输入。msg是提示用户的信息，resp是用户输入的数据。

我们先看一下基于GTK+实现的用户名和密码输入。

**实现接口函数**

```
static int show_prompt_dialog(int num_msg, const struct pam_message **msgm,
struct pam_response **response)
{
        int ret = 0;
        char* input = NULL;
        GtkWidget *table1;
        GtkWidget *label_prompt;
        GtkWidget *entry_input;
        GtkWidget *hbox1;
        GtkWidget *hbox2;
        GtkWidget *button2;
        GtkWidget *hbox3;
        GtkWidget *button3;
        GtkWidget *hbox4;
        gboolean need_input = msgm[0]->msg_style == PAM_PROMPT_ECHO_OFF
                        || msgm[0]->msg_style == PAM_PROMPT_ECHO_ON;

        GtkDialog *dialog = (GtkDialog*)gtk_message_dialog_new(NULL, GTK_DIALOG_MODAL,
                need_input ? GTK_MESSAGE_QUESTION : GTK_MESSAGE_INFO,
                need_input ? GTK_BUTTONS_OK_CANCEL: GTK_BUTTONS_CLOSE,
                msgm[0]->msg);
```

```
        gtk_window_set_title (GTK_WINDOW (dialog), _("Please input the password!"));
        gtk_window_set_position(GTK_WINDOW(dialog), GTK_WIN_POS_CENTER_ALWAYS);
        table1 = gtk_table_new (3, 2, FALSE);
        gtk_widget_show (table1);
        gtk_box_pack_start (GTK_BOX (GTK_DIALOG (dialog)->vbox), table1, FALSE, FALSE, 0);

        if(need_input)
        {
            entry_input = gtk_entry_new ();
            gtk_widget_show (entry_input);
            gtk_table_attach (GTK_TABLE (table1), entry_input, 1, 2, 1, 2,
            (GtkAttachOptions) (GTK_EXPAND | GTK_FILL),
            (GtkAttachOptions) (0), 8, 8);

            if(msgm[0]->msg_style == PAM_PROMPT_ECHO_OFF)
            {
                gtk_entry_set_visibility (GTK_ENTRY (entry_input), FALSE);
                gtk_entry_set_invisible_char (GTK_ENTRY (entry_input), 8226);
            }
        }

    gtk_widget_hide(gtk_widget_get_parent(GTK_MESSAGE_DIALOG(dialog)->image));
    gtk_widget_hide(gtk_widget_get_parent(GTK_MESSAGE_DIALOG(dialog)->label));
    gtk_widget_show (GTK_WIDGET(dialog));
    ret = gtk_dialog_run(dialog);
    input = (char*)gtk_entry_get_text(GTK_ENTRY(entry_input));
    input = input != NULL ? strdup(input) : input;
    gtk_widget_destroy(GTK_WIDGET(dialog));

    if(ret == GTK_RESPONSE_OK)
    {
        struct pam_response * reply = (struct pam_response *) calloc(num_msg,
            sizeof(struct pam_response));

        if (reply != NULL)
        {
            reply[0].resp_retcode = 0;
            reply[0].resp = input;

            *response = reply;
        }

        setenv("USER_ACTION", "OK", 1);
        return PAM_SUCCESS;
    }
    else
    {
        free(input);

        setenv("USER_ACTION", "CANCEL", 1);
        return PAM_ABORT;
    }
}
```

```
static int authentication_conv(int num_msg, const struct pam_message **msgm,
        struct pam_response **response, void *appdata_ptr)
{
        int count = 0;

        if (num_msg != 1 || msgm == NULL || response == NULL)
        {
            return PAM_CONV_ERR;
        }

        return show_prompt_dialog(num_msg, msgm, response);
}
```

### 调用PAM进行鉴权

```
AuthVerifyPasswdResult authentication_verify_password(struct pam_conv* conv)
{
        int res = 0;
        char *user = NULL;
        pam_handle_t *pamh = NULL;

        uid_t uid = geteuid();
        struct passwd* pw = getpwuid(uid);

        if (pw)
        {
            user = pw->pw_name;
        }
        else
        {
            g_debug("Invalid userid: %lu\n", (unsigned long) uid);
            return FALSE;
        }

        pam_start("login", user, conv, &pamh);
        pam_set_item(pamh, PAM_TTY, "/dev/tty");
        res = pam_authenticate(pamh, 0);
        pam_end(pamh, res);

        if(res == PAM_SUCCESS)
        {
            res = AUTH_VERIFY_PASSWD_OK;
            g_debug("%s is ok: %s\n", __func__, user);
        }
        else
        {
            const char* user_action = getenv("USER_ACTION");
            if(user_action != NULL && strcasecmp(user_action, "CANCEL") == 0)
            {
                g_debug("%s is canceled: %s\n", __func__, user);
                res = AUTH_VERIFY_PASSWD_CANCEL;
            }
            else
```

```
        {
            g_debug("%s is fail: %s\n", __func__, user);
            res = AUTH_VERIFY_PASSWD_FAIL;
        }
        unsetenv("USER_ACTION");
    }

    return res;
}

AuthVerifyPasswdResult authentication_verify_password_simple(void)
{
    static struct pam_conv conv =
    {
        authentication_conv,
        NULL
    };

    return authentication_verify_password(&conv);
}
```

再看一下基于终端实现的用户名和密码输入。

**实现接口函数**

```
int misc_conv(int num_msg, const struct pam_message **msgm,
              struct pam_response **response, void *appdata_ptr)
{
    int count=0;
    struct pam_response *reply;

    if (num_msg <= 0)
    return PAM_CONV_ERR;

    D(("allocating empty response structure array."));

    reply = (struct pam_response *) calloc(num_msg,
                        sizeof(struct pam_response));
    if (reply == NULL) {
    D(("no memory for responses"));
    return PAM_CONV_ERR;
    }

    D(("entering conversation function."));

    for (count=0; count < num_msg; ++count) {
    char *string=NULL;
    int nc;

    switch (msgm[count]->msg_style) {
    case PAM_PROMPT_ECHO_OFF:
        nc = read_string(CONV_ECHO_OFF,msgm[count]->msg, &string);
        if (nc < 0) {
```

```
            goto failed_conversation;
            }
            break;
        case PAM_PROMPT_ECHO_ON:
            nc = read_string(CONV_ECHO_ON,msgm[count]->msg, &string);
            if (nc < 0) {
            goto failed_conversation;
            }
            break;
case PAM_ERROR_MSG:
            if (fprintf(stderr,"%s\n",msgm[count]->msg) < 0) {
            goto failed_conversation;
            }
            break;
        case PAM_TEXT_INFO:
            if (fprintf(stdout,"%s\n",msgm[count]->msg) < 0) {
            goto failed_conversation;
            }
            break;
        case PAM_BINARY_PROMPT:
        {
            pamc_bp_t binary_prompt = NULL;

            if (!msgm[count]->msg || !pam_binary_handler_fn) {
            goto failed_conversation;
            }

            PAM_BP_RENEW(&binary_prompt,
                 PAM_BP_RCONTROL(msgm[count]->msg),
                 PAM_BP_LENGTH(msgm[count]->msg));
            PAM_BP_FILL(binary_prompt, 0, PAM_BP_LENGTH(msgm[count]->msg),
                 PAM_BP_RDATA(msgm[count]->msg));

            if (pam_binary_handler_fn(appdata_ptr,
                        &binary_prompt) != PAM_SUCCESS
            || (binary_prompt == NULL)) {
            goto failed_conversation;
            }
            string = (char *) binary_prompt;
            binary_prompt = NULL;

            break;
        }
        default:
            fprintf(stderr, _("erroneous conversation (%d)\n"),
            msgm[count]->msg_style);
            goto failed_conversation;
}
        if (string) {                              /* must add to reply array */
            /* add string to list of responses */

            reply[count].resp_retcode = 0;
            reply[count].resp = string;
            string = NULL;
```

```
        }

        *response = reply;
        reply = NULL;

        return PAM_SUCCESS;

failed_conversation:

        D(("the conversation failed"));

        if (reply) {
        for (count=0; count<num_msg; ++count) {
            if (reply[count].resp == NULL) {
            continue;
            }
            switch (msgm[count]->msg_style) {
            case PAM_PROMPT_ECHO_ON:
            case PAM_PROMPT_ECHO_OFF:
            _pam_overwrite(reply[count].resp);
            free(reply[count].resp);
            break;
            case PAM_BINARY_PROMPT:
        {
            void *bt_ptr = reply[count].resp;
            pam_binary_handler_free(appdata_ptr, bt_ptr);
            break;
        }
            case PAM_ERROR_MSG:
            case PAM_TEXT_INFO:
            /* should not actually be able to get here... */
            free(reply[count].resp);
            }
            reply[count].resp = NULL;
        }
        /* forget reply too */
        free(reply);
        reply = NULL;
        }

        return PAM_CONV_ERR;
    }
(libpam_misc/misc_conv.c)
```

### 调用PAM进行鉴权

```
static struct pam_conv conv =
{
            misc_conv,
            NULL
};
authentication_verify_password(&conv);
```

（authentication_verify_password在前面已经实现过。）

上面的代码可能有些复杂，不用研究里面的细节，重在理解它的原理。在这个例子里，既实现了基于图形用户界面输入用户名和密码的方式，也实现了基于终端输入用户和密码的方式，两者的实现几乎没有重复的代码，这就证实了它的用户界面和内部实现分离得比较彻底。

## 11.2 MVC架构

在程序员中，MVC架构可谓家喻户晓了，它无疑是最负盛名的架构模式之一。可惜的是教科书里很少提及它，所以一些新手，特别是从其他专业转行过来的新手，对MVC架构了解不多。我见过的大部分应届毕业生，要么对MVC不了解，要么存在一些重大误解，最多只是能硬记得一些概念而已。

Trygve Reenskaug最初在1979年提出MVC架构，到现在已经30年的历史了。这里说句题外话，软件行业无疑是发展最快的行业之一，不少高人提出过类似的结论：由于知识更新太快，做程序员必须不断地学习，否则很快会被淘汰。做程序员必须不断地学习，这是对的，其实这也是做程序员的乐趣之一，不要把学习当作一项任务来完成，否则你最好不要进入这个行业。不过，知识更新快并不是促使我们学习的主要原因，实际上，大部分经典的理论和方法，像数据结构、离散数学、编译原理、操作系统原理和面向对象设计等等，在过去几十年里都没有大的变化（它们确实在不断完善），大部分所谓的“新”技术，无非是一些商业公司的噱头而已。初学者应该把主要精力放在学习经典的理论和方法上，而不是一味的追逐“新”技术，否则除了知道几个新概念外，不会学习到实质性的东西，也不能静下心做点实事。

言归正传，MVC架构是让软件系统模块化的一种方法，是前面所讲的分层架构的一个特例，它把软件系统划分成如图11-8所示的三大部分。

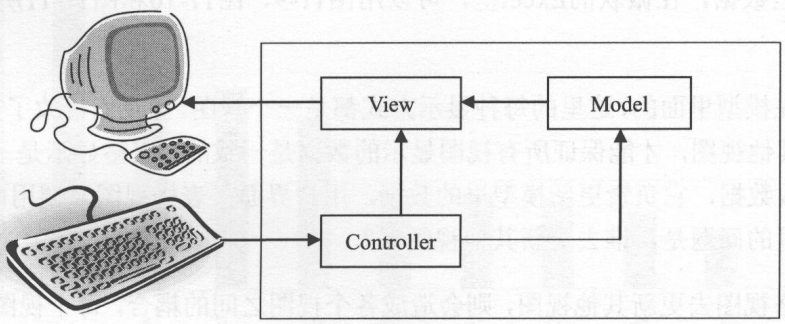



图11-8 MVC架构

**模型**（Model）：M是Model的首字母缩写。它负责实现程序的具体功能，包括核心数据结构和逻辑处理。也就是本章所说的内部实现。

**视图**（View）：V是View的首字母缩写。它负责向用户显示处理结果。是本章所说的用户界面的一部分。

**控制器**（Controller）：C是Controller的首字母缩写。它接收用户的输入，然后调用模型的处理函数进行处理。是本章所说的用户界面的一部分。

视图和控制器合起来组成本章所说的用户界面。用户界面包括输入和输出两部分，视图相当于输出部分（即显示结果给用户），控制器相当于输入部分（即响应用户的操作）。

在大部分系统中，控制器的角色都被忽略了，或者作为视图的一部分对待了。在《企业应用架构模式》[①]中，Fowler[②]先生说过："视图和控制器的分离，就不那么重要了。实际上，几乎Smalltalk的每个版本都没有让视图和控制器的实现分离。为什么你会想分离它们呢，经典的例子是支持可编辑和不可编辑的行为。可以在这两种情况下，用一个视图和两个控制器的实现，这里控制器是视图的策略（strategies）。"在后面的示例中，我们不会特别强调控制器，提及视图的输入功能时指的就是MVC架构中的控制器。

MVC架构是分层架构的特例，它的主要特征是同一个模型可以支持多个视图。比如对于如表11-1所示的2009年7月编程语言排行榜的数据（数据源于CSDN）。

**表11-1　2009年7月的编程语言排行榜**

| 语　言 | 所占比例 | 语　言 | 所占比例 |
|---|---|---|---|
| Java | 20.45% | Python | 4.44% |
| C | 17.32% | Perl | 4.20% |
| C++ | 10.42% | Javascript | 3.51% |
| PHP | 9.27% | Ruby | 2.57% |
| Basic | 7.79% | Delphi | 2.00% |
| C# | 4.54% | 其他 | 13.50% |

同样是这些数据，在微软的Excel里，可以用图11-9、图11-10和图11-11所示的多种方式来显示。

数据是放在模型里面的，这里的每种显示方式都是一个视图。当我们修改了其中某项数据时，需要更新所有其他视图，才能保证所有视图显示的数据是一致的。表格显示是一个视图，我们在表格视图上修改数据，它负责更新模型里的数据，用户界面（表格视图）调用内部实现（模型）是合理的。这里的问题是，谁去更新其他视图呢？

如果由表格视图去更新其他视图，则会造成各个视图之间的耦合，每个视图都需要知道其他

① 已由机械工业出版社出版。——编者注

② Martin Fowler：软件开发方面的大师级人物，他在面向对象的分析设计、UML、模式、软件开发方法学、极限编程、重构等方面都是世界顶级的专家，现为ThoughtWorks公司的首席科学家。——编者注

视图的存在，删除一个视图或增加一个视图会遇到麻烦。

| | A | B | C | D | E |
|---|---|---|---|---|---|
| 1 | Java | 20.45% | | | |
| 2 | C | 17.32% | | | |
| 3 | C++ | 10.42% | | | |
| 4 | PHP | 9.27% | | | |
| 5 | Basic | 7.79% | | | |
| 6 | C# | 4.54% | | | |
| 7 | Python | 4.44% | | | |
| 8 | Perl | 4.20% | | | |
| 9 | Javascript | 3.51% | | | |
| 10 | Ruby | 2.57% | | | |
| 11 | Delphi | 2.00% | | | |
| 12 | 其他 | 13.50% | | | |
| 13 | | | | | |
| 14 | | | | | |
| 15 | | | | | |

图11-9 表格显示

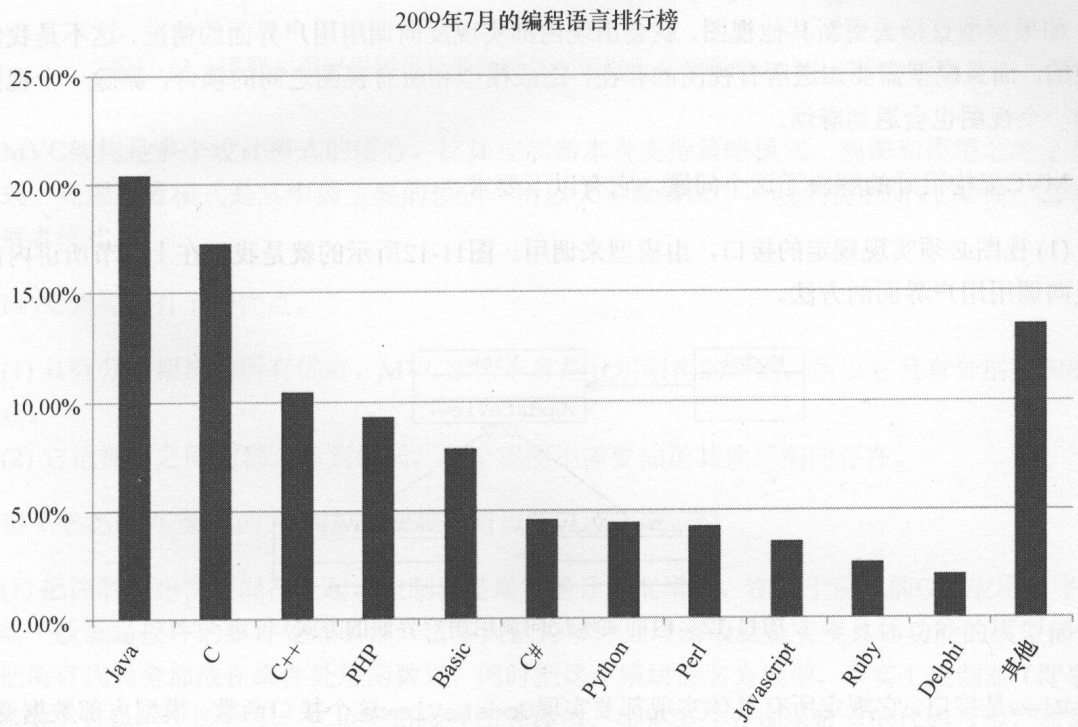



图11-10 柱状图显示

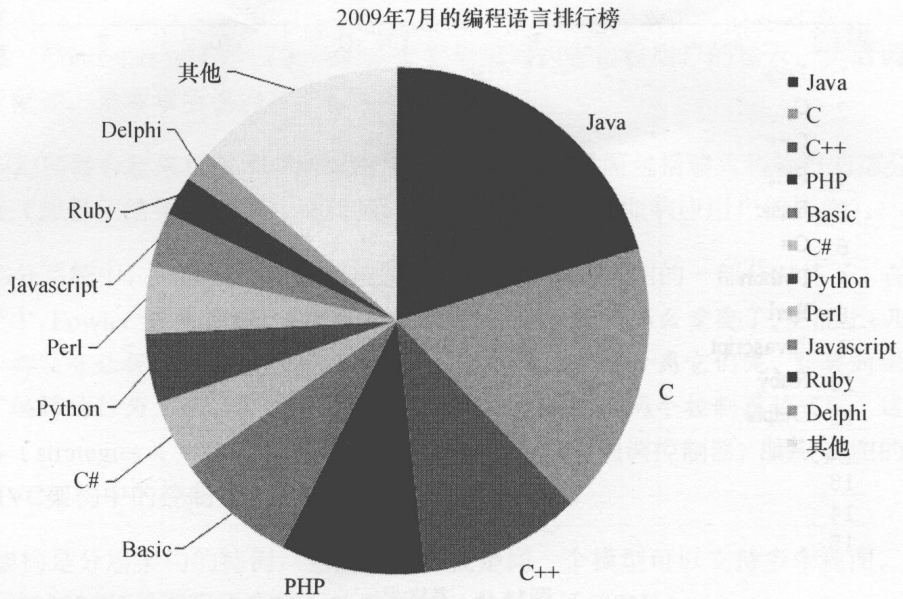



图11-11　饼状图显示

如果模型直接去更新其他视图，就会出现内部实现反向调用用户界面的情况，这不是我们所期望的。而且模型需要知道所有视图的存在，造成模型和所有视图之间的耦合，删除一个视图或增加一个视图也会遇到麻烦。

MVC架构很好的解决了这个问题，它有以下要求。

(1) 视图必须实现规定的接口，由模型来调用。图11-12所示的就是我们在上一节所讲内部实现反向调用用户界面的方法。

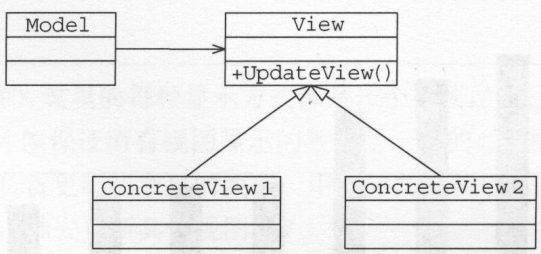



图11-12　内部实现反向调用用户界面的方式

View是接口，它规定所有具体实现都要实现UpdateView这个接口函数。模型内部数据变化，需要更新用户界面时，它就调用视图的UpdateView函数。由于视图接口是抽象的，模型调用视图接口不造成内部实现和用户界面的耦合。

(2) 在初始化时，所有视图都必须向模型注册。

(3) 模型内部数据变化，需要更新用户界面时，它依次调用所有视图的UpdateView函数，如图11-13所示。

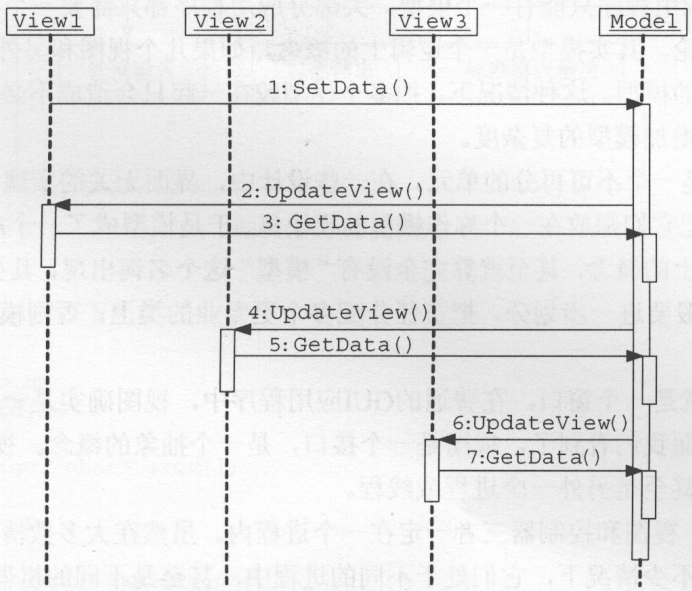



图11-13 模型在需要更新用户界面时依次调用所有视图的UpdateView函数

MVC架构是多个设计模式的组合，比如控制器本身支持策略模式，视图和模型之间是观察者模式。而观察者模式是其中最主要的模式，所以大多数情况下，我们提到MVC架构，通常是指观察者模式。

MVC架构具有下列优点。

(1) 具有分层架构的所有优点。MVC架构本身是分层架构的特例，所以它具有分层架构的所有优点。

(2) 它把视图之间的耦合降到最低，各个视图不需要知道其他视图的存在。

但不熟悉MVC架构的人对MVC架构常有以下几点误解。

(1) 把控制器和模型混在一起。控制器是最容易让人混淆的，在基于窗口的GUI应用程序中，控制器一般就是控件的事件处理函数，很多人认为事件处理函数就是实现具体功能的模型部分，因此把所有代码全部放在事件处理函数里，同时把这个模块命名为模型。事实上控制器（即事件处理函数）只是一个胶合层，它负责衔接视图和模型。如果事件处理函数里的代码过多，通常就意味着控制器和模型被混在一起了。

(2) 把所有用户界面都与模型关联。有的用户界面是比较独立的，不存在内部实现反向调用

用户界面的情况，完全能以一种调用关系来实现，没有必要与模型挂钩。把所有用户界面都与模型关联的做法看似充分利用了模型，实际上只是把模型当成了全局变量的替代品。结果既增加了用户界面与模型耦合，让这些独立的界面难以重用，也增加了模型复杂度。

(3) 认为一个应用程序只能有一个模型。大部分应用程序都只需要一个模型，但一些人把这个现象这看成了定论。其实模型是一个逻辑上的概念，如果几个视图和另外几个视图毫无关联，最好各自使用独立的模型。这种情况下，把多个模型绞在一起只会造成不必要的耦合，使代码重用变得困难，也让增加模型的复杂度。

(4) 认为模型是一个不可再分的单元。在一些设计中，界面无关的实现代码的确与用户界面代码分开了，却又把它们都放在一个称作模型的模块里，于是模型成了一个庞然大物。我们说了，模型只是一个逻辑上的概念，甚至就算完全没有“模型”这个名词出现，其架构仍然是基于MVC的。模型的功能一般要进一步划分，把责任分配多个更专业的类上。否则模型会过于复杂，使实现的难度增加。

(5) 认为视图就是一个窗口。在普通的GUI应用程序中，视图确实是一个窗口或子窗口，但这并不是定论。前面我们看到了，视图是一个接口，是一个抽象的概念。视图可能是一个窗口，也可能是个终端，甚至是另外一个进程或线程。

(6) 认为模型、视图和控制器三者一定在一个进程内。虽然在大多数情况下，这三者确实在一个进程里。但在不少情况下，它们处于不同的进程中，甚至是不同的机器上。

## 11.3 外壳模式

如果一个应用程序已经写好了，而且它的用户界面和内部实现并没有很好的分离。出于某些原因，你不想或不能去修改原有的应用程序，但你又希望为它加上一个新的用户界面，这时，外壳（shell）模式或许可以帮助你。

之所以说是“或许可以”，原因是外壳模式它基于这样一个假设：应用程序实现了基于终端的用户界面，即从标准输入（stdio）中读取数据，向标准输出（stdout）和标准错误输出（stderr）显示结果。标准输入、标准输出和标准错误输出都是文件描述符，文件描述符是一个整数，具有天生的抽象能力（Unix把硬件都当作文件来处理）。

我们把标准输入、标准输出和标准错误输出重定向到管道上，向管道里写数据来模拟应用程序的输入，从标准输出和标准错误输出里读取应用程序的输出。这样一来，就不需要修改原来的应用程序，而控制它的输入和输出(即用户界面)，同时应用程序也不知道外壳的存在。

外壳模式的组成如图11-14所示。

我们先看一个DEMO程序，它演示了外壳模式的基本原理。

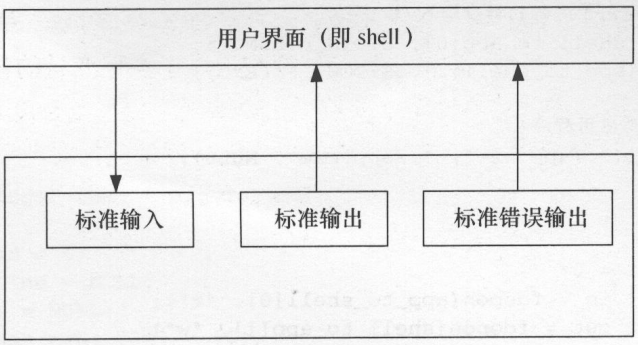



图11-14 外壳模式

应用程序（app.c）如下所示。

```
#include <stdio.h>
#include <unistd.h>

int main(int argc, char* argv[])
{
        int n = 0;

        printf("Input number:\n");
        fflush(stdout);
        scanf("%d", &n);
        printf("You input %d\n", n);

        return 0;
}
```

这个程序提示用户输入一个整数，然后打印出来。

现在我们给它加一个外壳（shell.c）。

```
#include <unistd.h>
#include <stdio.h>

int main(int argc, char* argv[])
{
        int shell_to_app[2] = {0};
        int app_to_shell[2] = {0};
        /*创建两个管道*/
        pipe(shell_to_app);
        pipe(app_to_shell);

        if(fork() == 0)
        {
            close(shell_to_app[1]);
            close(app_to_shell[0]);
```

```
        /*重定向子进程的标准输入/输出*/
        dup2(shell_to_app[0], STDIN_FILENO);
        dup2(app_to_shell[1], STDOUT_FILENO);

        /*启动应用程序*/
        execl("./app.exe", "./app.exe", NULL);
    }
    else
    {
        int i = 0;
        FILE* in = fdopen(app_to_shell[0], "r");
        FILE* out = fdopen(shell_to_app[1], "w");

        char message[256] = {0};
        close(shell_to_app[0]);
        close(app_to_shell[1]);
        /*通过管道与应用程序交互*/
        fgets(message, sizeof(message), in);
        printf("1: %s\n", message);
        fprintf(out, "1234\n");
        fflush(out);
        fgets(message, sizeof(message), in);
        printf("2: %s\n",  message);
        printf("2: %s\n",  message);
        fclose(in);
        fclose(out);
    }

    return 0;
}
```

不修改原有应用程序的代码，外壳为应用程序提供了一个新的用户界面。这里的app单独运行时，需要用户手工输入数据，而在外壳的帮助下，输入自动化了。

gdb是一个基于终端的调试器，它的功能强大使用方便，但是对初学者来说不太友好。为了方便喜欢图形用户界面的程序员，ddd（http://www.gnu.org/software/ddd/）为gdb提供了一个外壳。它实现了一个完全基于图形用户界面的调试器，它没有修改gdb的代码，gdb也不知道它的存在。基于外壳模式，它以一种巧妙的方式为gdb提供了一个新的用户界面。实际上gdb还有其他一些外壳，比如集成到vim和emacs等编辑器里的gdb外壳。

这里我们为jdb实现一个外壳，jdb是Java的命令行调试器，它的功能并不弱于基于IDE的Java调试器。不过和gdb相比，它实在太难用了：没有命令的历史记录，不支持命令自动补全，不支持命令缩写。

为了改进jdb的可用性，我们给它加一个新的外壳，这个外壳同样是基于终端的，但是支持上面提到的这些功能。

readline是一个用于提高基于终端的应用程序可用性的函数库，它能提供命令的历史记录

和命令自动补全等功能（gdb也是基于readline实现的）。这里我们利用readline为jdb添加上述功能。

### 主函数

```
int main (int argc, char **argv)
{
        int  pid = 0;
        char *line = NULL;
        char *s = NULL;
        char last_cmd[1024] = {0};
        int  parent_to_child[2] = {0};

        if(argc == 1)
        {
            show_usage();
            return 0;
        }

initialize_readline ();
/*创建一个管道*/
        pipe(parent_to_child);

        pid = fork();
        if(pid == 0)
{
        /*重定向jdb的标准输入*/
            close(parent_to_child[1]);
            dup2(parent_to_child[0], STDIN_FILENO);
            strcpy(argv[0], "jdb");

            /*启动jdb*/
            int ret = execvp("jdb", argv);
            fprintf(stderr, "jdb exit: ret=%d\n", ret);
        }
        else
{
            close(parent_to_child[0]);
            /*调用readline接收用户输入，做适当处理之后，转发给jdb*/
            while(1)
            {
                int i = 0;
                line = readline ("");

                if (!line) break;

                s = stripwhite (line);
                /*记录命令到历史记录*/
                if(*s)
                {
                add_history (s);
                strncpy(last_cmd, s, sizeof(last_cmd)-1);
```

```
            }
            else if(last_cmd[0])
            {
            /*回车键重复上次的命令(和gdb一样)*/
            s = last_cmd;
            printf(">%s\n", s);
            }
            else
            {
            free (line);
            printf(">");
            continue;
            }
            /*查询缩写的命令*/
            for(i = 0; g_alias[i].alias != NULL; i++)
            {
            int len = strlen(g_alias[i].alias);
            if(strncmp(g_alias[i].alias, s, len) == 0
                && (s[len] == ' ' || s[len] == '\0' || s[len] == '\t'))
            {
                /*把命令转发给jdb*/
                write(parent_to_child[1], g_alias[i].command,
                    strlen(g_alias[i].command));
                break;
            }
            }

            if(g_alias[i].alias == NULL)
            {
            /*把命令转发给jdb*/
            write(parent_to_child[1], s, strlen(s));
            }

            if(write(parent_to_child[1], "\n", 1) <= 0)
            {
            fprintf(stderr, "jdb exited\n");
            break;
            }
            free (line);
        }

        kill(pid, SIGTERM);
    }

    return 0;
}
```

### 命令自动补全

(1) 定义支持自动补全的命令列表。

```
static const char* commands[] =
{
```

```
        "class",
        "classes",
        "classpath",
        "clear",
        "connectors",
        "cont",
        "disablegc",
        "down",
        "dump",
        "enablegc",
        "eval",
        "exit",
        "fields",
        "help",
        "interrupt",
"kill",
...
        "wherei",
        NULL
};
```

(2) 匹配命令补全的函数。

```
char* command_generator(const char *text, int state)
{
        const char *name;
        static int list_index, len;

        if (!state)
        {
            list_index = 0;
            len = strlen (text);
        }

        while (name = commands[list_index])
        {
            list_index++;

            if (strncmp (name, text, len) == 0)
                return strdup(name);
        }

        return ((char *)NULL);
}
```

**实现readline需要的回调函数**

```
char** command_completion (const char *text, int start, int end)
{
        char **matches = NULL;

        if (start == 0)
            matches = rl_completion_matches (text, command_generator);
```

```
        return (matches);
}
```

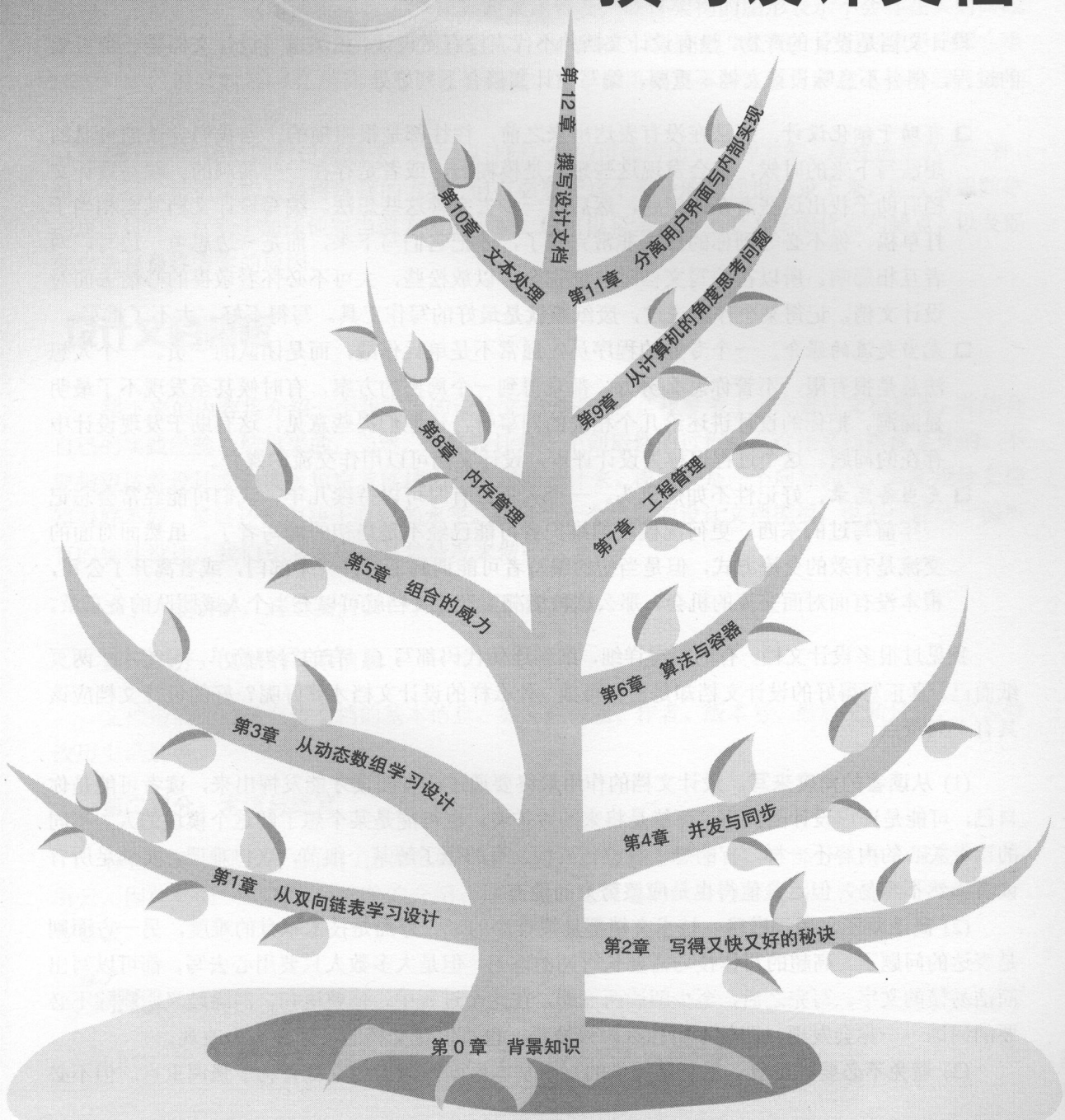
**初始化readline**

```
void initialize_readline ()
{
        rl_readline_name = "jdbshell";
        rl_attempted_completion_function = command_completion;

        return ;
}
```

外壳模式可以作为补救方案来改进现有的应用程序，也可以用这个模式来设计新的应用程序。如果使用恰当，它会为我们带来极大的方便。

# 第 12 章

# 撰写设计文档


第12章 撰写设计文档
第10章 文本处理
第11章 分离用户界面与内部实现
第9章 从计算机的角度思考问题
第8章 内存管理
第7章 工程管理
第5章 组合的威力
第6章 算法与容器
第3章 从动态数组学习设计
第4章 并发与同步
第1章 从双向链表学习设计
第2章 写得又快又好的秘诀
第0章 背景知识

设计是软件开发中的一个重要环节，设计的好坏直接影响软件的质量和开发的工作量。它是软件开发的必经阶段，不管程序规模的大小，都少不了这个过程。对于小程序，你在脑子稍微想一想，就知道怎么做了，这个想的过程就是设计的过程。对于大程序，你可能要花上几周甚至更久的时间去思考、和同事讨论，并做些试验去验证你的想法，最终才能确定应该怎么做。

设计文档是设计的产物，没有设计文档并不代表没有做设计。虽然编写设计文档是一个可选的过程，但并不意味设计文档不重要，编写设计文档有下列好处。

- 有助于细化设计。想法在没有表达出来之前，往往都是很粗糙的。当我们尝试把自己的想法写下来的时候，就会发现这些想法是模糊的，或者是存在一些漏洞的。编写设计文档有助于找出这些潜在的问题，然后进一步去完善这些想法。编写设计文档其实相当于打草稿，你不必等到你的想法非常完善了，才把它们写下来，而是一边思考一边写，两者互相影响。所以在编写文档时，你完全可以放松些，大可不必怀着敬畏的心情去面对设计文档。记得某个作家说过，废纸篓就是最好的写作工具。写得不好，大不了重写。
- 充当交流的媒介。一个专业的程序员，通常不是单兵作战，而是团队的一员。一个人想法总是很有限，不管你怎么努力，都难得到一个周密的方案，有时候甚至发现不了最明显漏洞。把你的设计讲述给几个相关的同事听，让他们提些意见，这有助于发现设计中存在的问题。这个过程也称为设计评审，设计文档可以用作交流的媒介。
- 充当备忘录。好记性不如烂笔头。一个大项目有时可以持续几年，我们可能经常会忘记一年前写过的东西，更何况模块的维护者可能已经不是当初的编写者了。虽然面对面的交流是有效的交流方式，但是当初的编写者可能调到了另外一个部门，或者离开了公司，根本没有面对面交流的机会。那么这种情况下设计文档就可以充当个人或团队的备忘录。

我见过很多设计文档，有的写得详细，细到连伪代码都写了，有的写得简单，可能才一两页纸而已。真正写得好的设计文档却是屈指可数。什么样的设计文档才算好呢？好的设计文档应该具有下列特点。

(1) **从读者的角度来写**。设计文档的作用最终要通过读者阅读才能发挥出来，读者可能是你自己，可能是评审设计的同事，可能是将来的维护者，也可能是某个想了解这个模块的人。不同的读者关心的内容不一样，有的想了解整体架构，有的想了解某个细节，众口难调。要满足所有读者当然不容易，但这是值得也是应该努力的地方。

(2) **简洁明了，用词准确**。技术文档都是很难读的，一方面是技术本身的难度，另一方面则是表达的问题了。高超的写作技巧需要长时间的练习，但是大多数人只要用心去写，都可以写出简洁易懂的文字。写完之后，至少朗读两三遍，在这个过程中，修整语句，消除歧义，删除不必要的词语……你会发现，即使不用什么修辞技巧，也可以让文档的可读性大为改观。

(3) **避免不必要的重复**。重复是美学的一个基本原则，适当的重复有利于强调重点。但不必

要的重复会让读者感到迷惑，读者会怀疑自己是不是看错了，或者去思考里面还有什么其他玄妙。不必要的重复也会让文档维护变得困难，当文档内容有变化时，你得修改所有重复的地方。如果忘掉一两处，也会让读者感到困惑。

(4) **使用标准的符号和术语，自定义的符号和术语要有明确解释**。表示软件架构的方法有很多种，幸运的是，通用建模语言（UML）越来越普及，软件架构的图形表示不会存在大的问题了。即使是手工绘制的UML图或用非正式工具（如Word）绘制的UML图，它们可能不是那么准确和标准，但是熟悉UML的读者都可以看得懂。对于术语则要多加注意，即使是标准的，如果不常见，也要加些说明。

(5) **记录结果的同时记录想法**。在做设计时，我们可能考虑了多种方案，最终选定其中一种。设计文档通常记录的是最终选择的方案，但有必要把这个思考的过程也记录下来，让读者跟随你的思考过程，有利于加深他对当前方案的理解。你可以记录几种主要方案的优点和缺点，以及促使你选择最终方案的原因。

## 设计文档模板

接下来的内容将介绍一个我常用的设计文档模板，它是我参考了多个设计文档模板，并结合自己的实践经验总结出来的。当然，软件设计是一个创造性的过程，而设计文档模板是死的，不要指望按着设计文档模板填空就能填出一个好的设计来。模板只是一个向导而已，其实设计文档写得好不好，最终还是取决于自己的水平。就算是一个很好的设计文档模板，也不可能适于的所有的软件设计，我们还需要根据具体情况来做些取舍。

为了方便讲解，这里我们以一个歌词解析器为例。

**第一部分：文档的基本信息**

文档最前面应该有该文档的基本信息，如文档标题、作者、版本号、最后更新日期，以及修改历史记录等。

**第二部分：术语与缩写**

有些术语是比较专业的，比如多媒体编解码方面的术语和缩写，大部分人看了可能会不知所云，因此前面一定要有个简单的介绍。这个部分是可选的，有内容就写，没有也不用硬加一些进去。


LRC文件：一种描述歌词的文本文件，用于播放器同步显示歌词。

**第三部分：背景**

对背景的描述并非必须，视具体情况加以阐述。有些东西，设计者本人比较清楚，而文档读者可能知之甚少，若不加以说明就出现在后面的设计中，会让文档读者莫名其妙。比如说采用DBUS作为进程间通信机制，采用SQLite作为数据库管理系统，对这些技术做些简单的说明，并提供一些参考资料，有助于新手理解该设计文档。


LRC歌词文件是一种描述歌词的文本文件，用于播放器同步显示歌词。它由三种类型的数据单元组成。

**时间标签（Time tag）**：格式为[mm:ss]或[mm:ss.fff]（分钟数:秒数）。数字必须为十进制的非负整数，比如[12:34.5]是有效的，而[0x12:-34.5]是无效的。时间标签可以位于歌词文件的任意位置。一行歌词可以包含多个时间标签（比如歌词中的迭句部分）。根据这些时间标签，应用程序（如播放器）会按顺序依次高亮当前的歌词，从而实现卡拉OK功能。时间标签并不一定按时间排序，排序功能由歌词解析程序来处理。

**标识标签（ID tag）**：格式为[标识名:值]，标识名对大小写不敏感。以下是预定义的标签。

```
[ar:艺人名]
[ti:曲名]
[al:专辑名]
[by:本文作者]
[offset:时间补偿值] 其单位是毫秒，正值表示整体提前，负值相反。这是用于总体调整显示快慢的。
```

**歌词**：这就是实际的歌词了，通常位于时间标签之后，在解析时歌词可以作为时间标签的一部分对待。

如

```
[ti:I have nothing]
[ar:Whitney Houston]

I Have Nothing
Whitney Houston
[00:20.95]Share my life, take me for what I am
[00:30.47]Cause I'll never change all my colours for you
[00:40.16]Take my love, I'll never ask for too much
[00:49.09]Just all that you are and everything that you do
[02:42.91][00:58.78]I don't really need to look very much further
[02:47.74][01:03.69]I don't want to have to go where you don't follow
[02:52.37][01:08.25]I won't hold it back again, this passion inside
[02:56.62][01:12.40]Can't run from myself
```

```
[02:59.05][01:15.18]There's nowhere to hide
[03:03.02](Your love I'll remember forever)
[03:39.61][03:09.02][01:19.98]Don't make me close one more door
[03:46.85][03:13.73][01:24.92]I don't wanna hurt anymore
[03:51.05][03:17.90][01:29.04]Stay in my arms if you dare
[03:56.01][03:22.98][01:34.29]Or must I imagine you there
[04:09.89][04:06.04][04:00.35][03:27.61][01:38.77]Don't walk away from me...
[04:15.73][03:32.56][01:43.95]I have nothing, nothing, nothing
[04:21.27][01:49.16]If I don't have you, you
[01:59.65]you ,you, you
[04:30.64]If I don't have you
[02:05.71]You see through, right to the heart of me
[02:14.49]You break down my walls with the strength of you love
[02:24.11]I never knew love like I've known it with you
[02:32.69]Will a memory survive, one I can hold on to
[04:36.33]Oh~~
```

**第四部分：需求简述**

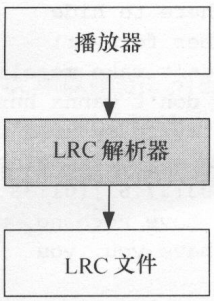
需求简述主要起承上启下的作用，上承SPCE（软件需求）中详细需求，下启设计文档中的各种解决方案。 大部分读者并不关心需求的细节以及对需求的理解，只要无碍于阅读后面的文档即可。让读者去阅读几十页的SPEC有些不近人情，这里的需求简述能为读者提供一些帮助。你可以采用散文式的叙述，也可以用UML中的用例图/场景来表示。


LRC歌词解析器的基本功能如下：
根据标识名查询标识名对的值；
查询指定时间对应的歌词；
按时间顺序遍历歌词；
针对嵌入式系统设计，要考虑到主流手机的可移植性以及硬件的能力。

**第五部分：与外部模块的交互**

多数模块都不是孤立的，而是存在于一个上下文之中的。用一张粗略但囊括全貌的图表来说明某个模块在系统中所处的位置，以及它与其他模块的交互关系，并配以简要的文字说明，可以免去读者在阅读时“不识庐山真面目”的迷茫和压抑。


应用程序，如多媒体播放器都可以调用LRC解析器来操作LRC文件。




**第六部分：分析**

设计是一个针对多个目标进行优化的过程，这些目标却往往是互相竞争的。实现功能当然是主要的方面，质量属性（性能、稳定性、扩展性和可测试性等）亦不能小觑。面对这些需求，你有哪些方案可供选择，在平衡各方面的因素后，你又选择了哪种方案？记下你的想法，为后面的设计提供理论基础。否则时间久了，你自己都忘了，到时候可能别人一句话就推翻了你的设计，你却是哑口无言。

**示例**

LRC解析器通常由下列几个部分组成。

**解析**。我们需要把平面的LRC数据，分解成一个一个的基本单元（比如各种标签），这个过程就是解析。由于LRC的格式稍微有点复杂，我们自然会想到解析文本数据的利器-状态机。所以我们将用状态机来解析LRC文件。

**表示**。为了操作方便，我们需要用一些数据结构来表示歌词。标识标签可以用一个结构来表示，由于标识标签有多个，所以需要一个容器来管理它们。标识标签的个数不多，我们可以用链表来管理。时间标签和歌词是关联的，共用一个数据结构来表示。时间标签也有多个，也需要一个容器来管理它们。时间标签个数也不多，通常只有几十个，我们也用链表来管理。这样，一个歌词文件由两个链表来表示，一个链表管理时间标签，一个链表管理标识标签。

**构建**。构建就是把解析器解析出来的基本单元转换成相应的数据结构。这可以由解析器来完成，一边解析一边构建。但那样会使解析器变得复杂，使解析器与构建之间的耦合太紧。这里我们采用Builder模式来分离解析和构建两个过程。

**操作**。为了让应用程序使用方便，我们还需要提供一些操作函数，用来查询歌词里的内容。

**其他方面的考虑**。文件操作函数和内存管理函数的可移植性。

**第七部分：对象模型**

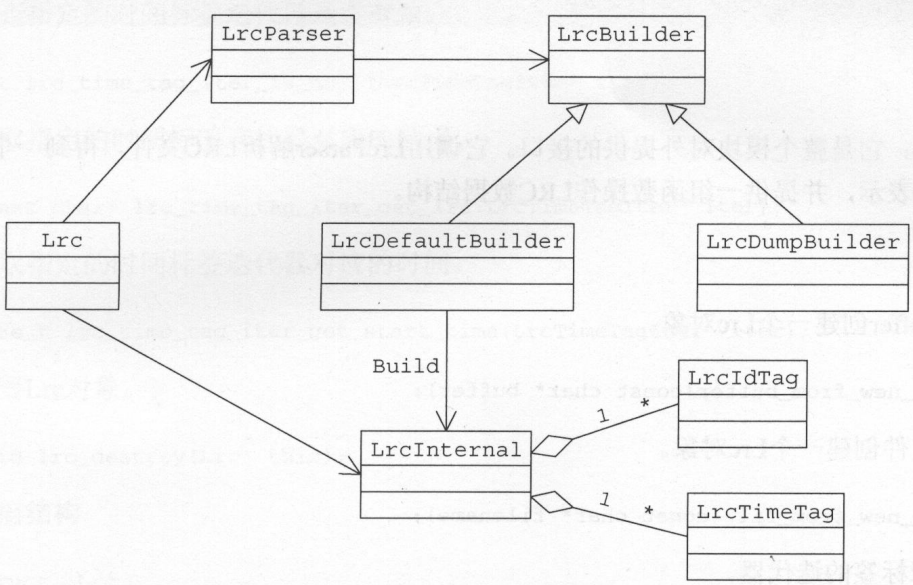
对象模型主要用于描述系统中有哪些类或子系统，以及它们之间的静态关系。这些关系包括继承关系，组合关系和关联关系等等。

(1) 总览

遵循自上而下的设计方法，利于自己的思考，也利于读者的理解。类图可以清析的表达出类/子系统之间的关系，这里可以用一张/多张类图或包图来表示系统的组成。在这里，类图中的类只要有一个描述性的名字即可，不用写得太细，否则反而起不到总览的效果。类的调用、组合、继承、分层等关系都在此描述清楚。


LrcBuilder是一个接口，它有LrcDefaultBuilder和LrcDumpBuilder两种实现。前者负责构建表示LRC的数据结构，后者主要用于打印调试信息。

LrcInternal是LRC的数据结构和内部操作，它由标识标签和时间标签两个链表组成，并提供一些基本的操作函数。




LrcParser负责LRC文件的解析，每解析到一个基本的单元就调用LrcBuilder的函数去构建。

Lrc是整个模块对外的提供的接口。它调用LrcParser解析LRC文件，得到一个LRC的内部数据结构表示，并提供一组函数操作LRC数据结构。

(2) 各个类/子系统的介绍

功能描述：模块存在的理由就是因为它承担了一定的职责，先在这里描述它的职责。

接口函数定义：模块为了实现它的职责，需要对外提供一组接口函数，这里一一进行描述（内

部函数则没有必要写在这里)。不可能开始就能把所有函数写出来，这需要一个不断完完善的过程。我习惯于先把头文件写出来，甚至把测试程序也写出来，这会刺激我的思考，能尽量完整的发现模块的接口函数。对于接口函数的描述包括以下内容(除了功能和名称外，其他都是可选的)：

功能；
名称；
返回值；
参数；
调用前提；
调用后的影响。

数据结构：最列出最重要的数据结构，不要事无巨细都写到这里。可能你还有必要写出这些数据结构存在的原因，毕竟它们不是无缘无故蹦出来的。


Lrc

**功能描述**：它是整个模块对外提供的接口。它调用LrcParser解析LRC文件，得到一个LRC的内部数据结构表示，并提供一组函数操作LRC数据结构。

**接口定义**

从一个buffer创建一个Lrc对象。

```
Lrc* lrc_new_from_buffer(const char* buffer);
```

从一个文件创建一个Lrc对象。

```
Lrc* lrc_new_from_file(const char* filename);
```

得到标识标签的迭代器。

```
LrcIdTagIter   lrc_get_id_tags(Lrc* thiz);
```

得到时间标签的迭代器。

```
LrcTimeTagIter lrc_get_time_tags(Lrc* thiz);
```

查询指定的标识标签，返回迭代器，可以通过迭代器获取相应的信息。

```
LrcIdTagIter   lrc_get_id_tag_get_by_key(Lrc* thiz, const char* key);
```

查询指定的时间标签，返回迭代器，可以通过迭代器获取相应的信息。

```
LrcTimeTagIter lrc_get_time_tag_by_time(Lrc* thiz, size_t start_time);
```

移动到指定时间标签的前一个时间标签。

```
LrcTimeTagIter lrc_time_tag_iter_prev(LrcTimeTagIter* iter);
```

移动到指定时间标签的下一个时间标签。

```
LrcTimeTagIter lrc_time_tag_iter_next(LrcTimeTagIter* iter);
```

检查指定的时间标签迭代器是不是第一个时间标签。

```
int lrc_time_tag_iter_has_prev(LrcTimeTagIter* iter);
```

检查指定的时间标签迭代器是不是最后一个时间标签。

```
int lrc_time_tag_iter_has_next(LrcTimeTagIter* iter);
```

检查指定的时间标签迭代器是否有效。

```
int lrc_time_tag_iter_is_null(LrcTimeTagIter* iter);
```

获取指定的时间标签迭代器对应的歌词。

```
const char* lrc_time_tag_iter_get_lrc(LrcTimeTagIter* iter);
```

获取指定的时间标签迭代器对应的时间。

```
size_t lrc_time_tag_iter_get_start_time(LrcTimeTagIter* iter);
```

销毁Lrc对象。

```
void lrc_destroy(Lrc* thiz);
```

**数据结构**

```
struct _Lrc
{
    LrcList* id_tags;
    LrcList* time_tags;
};
```

id_tags用来管理标识标签。
time_tags用来管理时间标签。

**第八部分：动态模型**

静态模型关注的是系统在设计时，各个类/子系统之间的静态关系。动态模型关注的是系统

在运行时，各个对象之间的协作与交互，状态的转换和数据的流动等动态行为。

(1) 总览

同理，这里要为文档的读者提供一个动态模型的概貌。

你采用的是管道过滤模型吗？

你采用的是MVC模型吗？

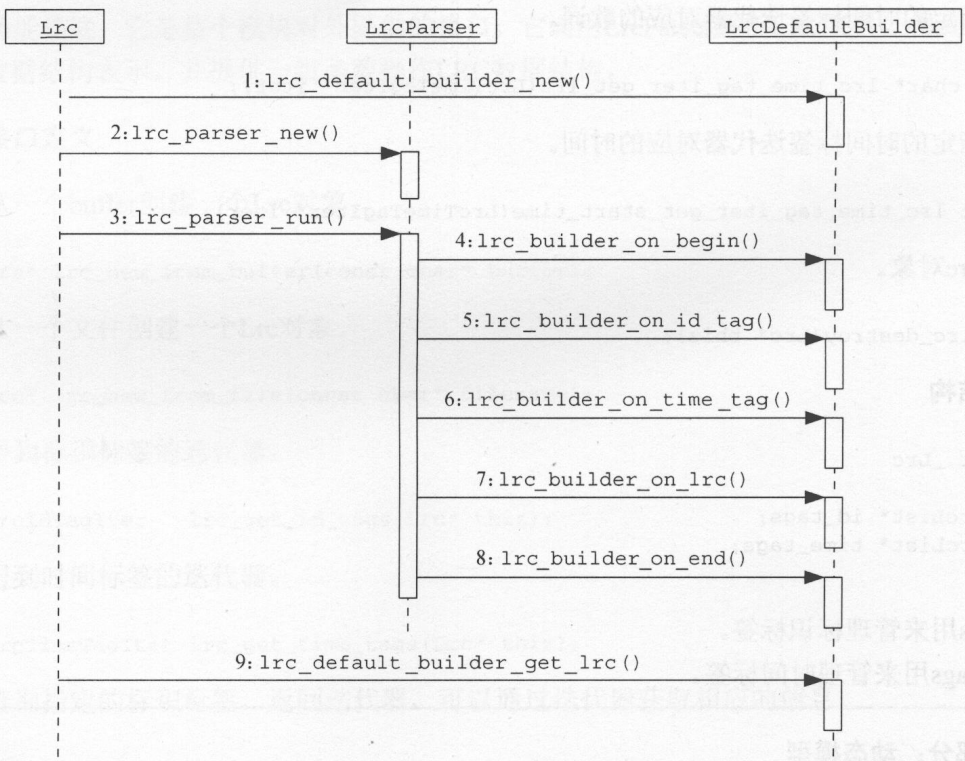
你采用的是共享数据模型吗？

你采用的是Builder模式吗？

……

如果你采用这些常见的模型作为系统的基本架构，有很多资料可以参考，那描述起来会方便很多。否则可能要多花些心思，才能将你的想法表达清楚。


LRC解析器采用Builder模式作为其基本架构，其主要交互如下：



(2) 交互关系

交互关系一般是针对特定使用场景的，常用UML中序列图或者活动图来表现。你可以列出一些典型的场景加以说明。或者对总览里的交互进行深入描述。

**示例**

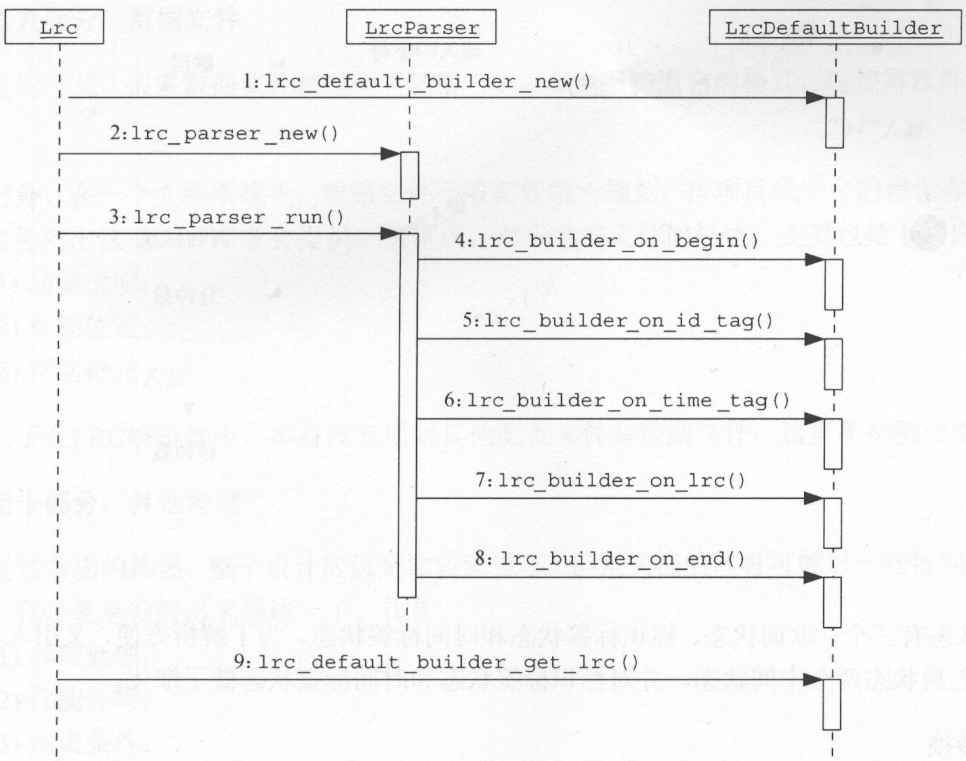
解析过程

这个过程比较简单。

先创建一个LrcDefaultBuilder对象，它负责构建Lrc的数据结构。

然后创建一个LrcParser对象，它负责Lrc数据的解析。

然后调用lrc_parser_run进行解析，传入LrcDefaultBuilder对象给它，它每解析到一个基本单元，如标识标签和时间标签，它就调用LrcDefaultBuilder对象的相应函数。

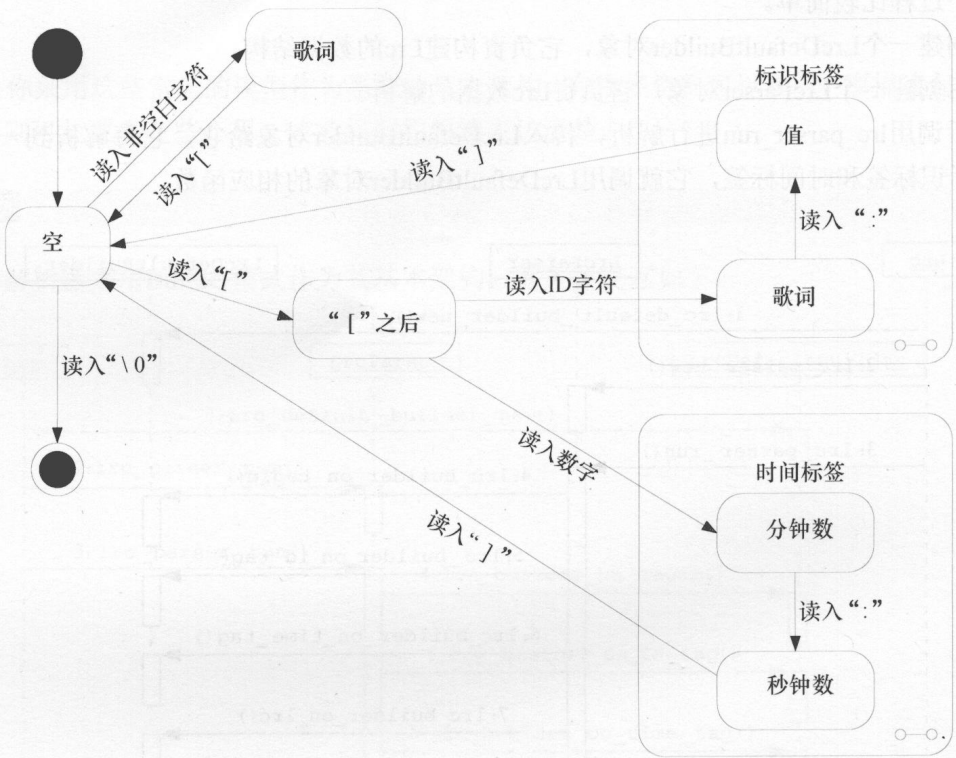



解析完成之后，从LrcDefaultBuilder对象中取出构建好的Lrc数据结构。

(3) 状态转换

交互关系侧重于对象之间的交互，状态转换则更强调对象自身的活动。尽管状态常常是在外部事件的驱动下才发生的，但状态本身都并不会从一个对象迁移到另外一个对象上。常用状态图来表现状态转换。


状态机是处理文本数据的利器，所以我们选择状态机来解析LRC文件，主要的状态转换可以用下图表示。




**状态**

主要状态有三个：歌词状态、标识标签状态和时间标签状态。为了解析方便，又引入了空状态和‘[’之后状态两个中间状态，并对标识标签状态和时间标签状态做了细化。

**状态转换**

在空状态下，读入‘[’时，进入‘[’之后状态。

在空状态下，读入非空白字符时，进入歌词状态。

在‘[’之后状态下，读入数字时，进入时间标签状态。

在‘[’之后状态下，读入ID字符时，进入标识标签状态。

在歌词状态下，读入‘[’时，返回到空状态。
在时间标签状态下，读‘]’时，返回空状态。
在标识标签状态下，读‘]’时，返回空状态。
在任何状态下，读入‘\0’时，进入结束状态。

(4) 数据流图

常用数据流图来表现某些系统，特别是基于数据处理的系统的功能。数据流图自然强调的是数据的流动，而交互图强调的是控制权的流动，数据流与控制流，两者呼应，对系统的描述更加全面。数据流图特别适合基于管道过滤器架构的系统。对于一般的系统来说，如果数据流不是特别重要，可以不用画它的数据流图。

由于LRC解析器中数据流并不重要，这里就不举例说明了，读者可以参考本书关于管道过滤器模式的一节。

**第九部分：数据文件**

时间一长，很多数据文件被时间“加密”了，谁也不知道它的格式，这使得软件的扩展受到限制。

另外，在一个大的系统中，数据文件一般都要统一规划，按项目或平台的惯例存放。把这些数据文件列出这里，评审者会提供一些建议。对于数据文件的描述，主要包括下列内容：

(1) 功能说明；
(2) 存储位置；
(3) 存储格式。

由于在LRC解析器中，本身没有用到其他配置文件和数据文件，这里不举例说明了。

**第十部分：其他考虑**

经过前面的描述，整个设计应该是比较清楚了。但是文档的读者可能对一些特别的问题比较关心，有必要单独提出来描述一下。比如：

(1) 异常处理；
(2) 性能分析；
(3) 约束条件；
(4) 可移植性。

**第十一部分：主要风险及未决事项**

当前模块可能依赖于硬件或者其他模块，依赖于其他一些测试结果之类的，列出这些风险和

未决事项，有助于跟踪。

**第十二部分：测试指南**

虽说当局者迷，但设计者从模块架构的角度出发，对它的脆弱之处可能看更清楚，掩盖它们只是掩耳盗铃。把它们写在这里，有助于功能完成之后的复查，也有利于测试者制定测试用例。

**第十三部分：设计回顾**

设计完成时，总结一下经验，对自己的设计方法加以反省和思考，有利于提高自己的设计能力，也能为他人提供供参考。如：

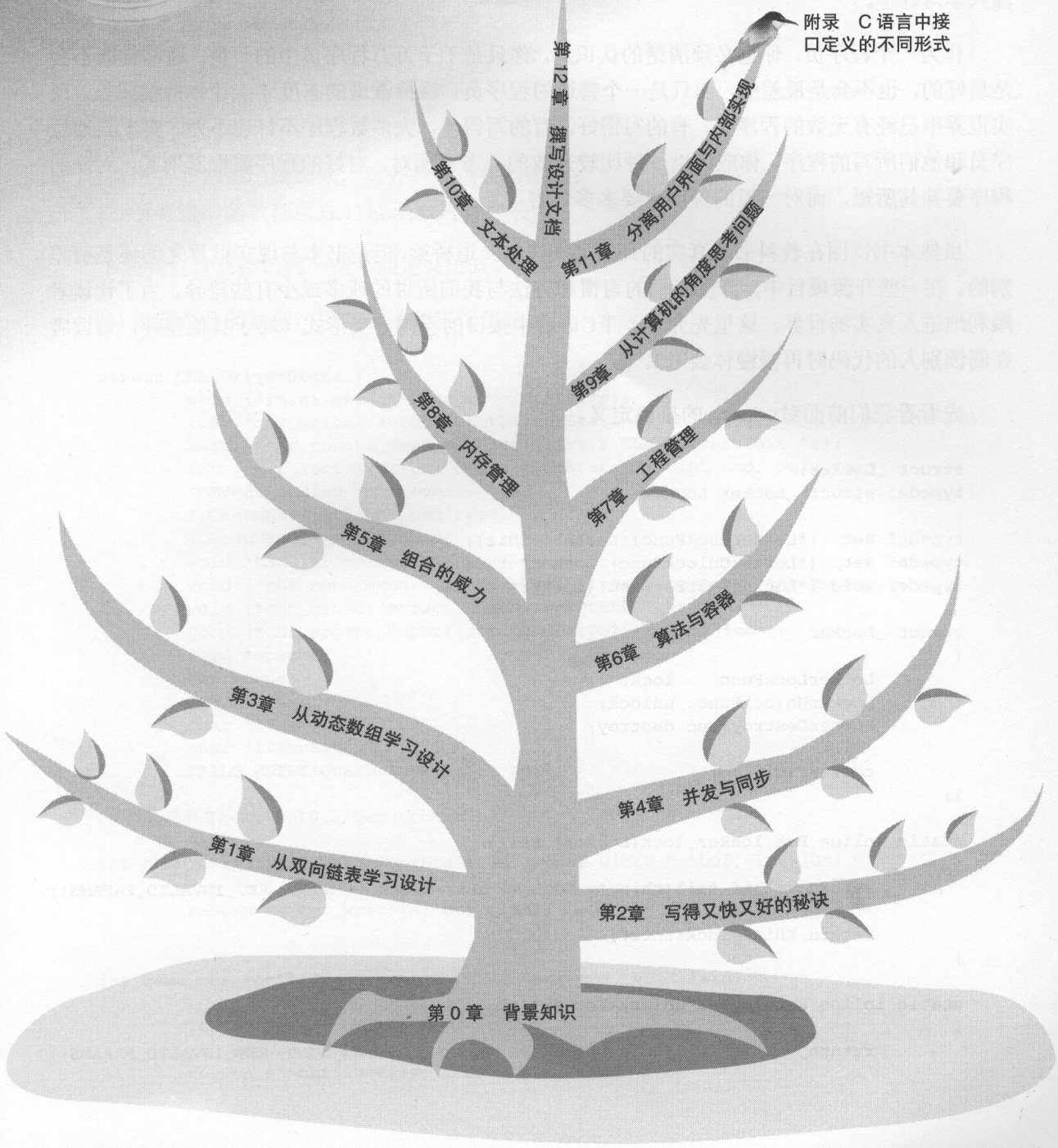
在设计过程遇到的问题以及解决方法；

设计时的一些心得；

对设计不太满意的地方，以及应该如何改进等。

# 附录

# C语言中接口定义的不同形式



如果你按照我在序言中所说的方法来阅读本书，并经过半年时间的努力学习，那么到那时，虽然你可能仍然不了解网络编程、不了解驱动程序、不了解3D图形开发，甚至连GUI程序都不会写，但是你已经无愧于程序员这个称号了。其他领域的知识，你需要学习、需要精通，这与你具体的发展方向有关，你并不需要掌握所有的东西，选择一两种（当然多多益善）其他方面的知识深入学习好了。

作为一个程序员，你也必须清楚的认识到，你只是千千万万程序员中的一员，通常你既不会是最好的，也不会是最差的，而只是一个普通的程序员，保持谦逊的态度才能让你持续成长。现实世界中已经有无数的程序了，有的写得好，有的写得差，大多数程序不好也不差。对于其他程序员和他们所写的程序，你应该以一种比较开放的心态去面对，对好的程序要取其所长，对差的程序要补其所短，而对一般的程序也要多多包容。

虽然本书试图在教科书和真实的开发之间建立一道桥梁，但是书本与现实世界之间还是有差别的。在一些开源项目中，开发者们的习惯和方法与我们所讲的或多或少有些差异。为了让读者顺利地进入真实的世界，这里先介绍一下C语言中接口的不同定义形式（对于其他差异，请读者在阅读别人的代码时再慢慢体会吧）。

先看看我们前面对Locker的接口定义。

```
struct _Locker;
typedef struct _Locker Locker;

typedef Ret  (*LockerLockFunc)(Locker* thiz);
typedef Ret  (*LockerUnlockFunc)(Locker* thiz);
typedef void (*LockerDestroyFunc)(Locker* thiz);

struct _Locker
{
        LockerLockFunc    lock;
        LockerUnlockFunc  unlock;
        LockerDestroyFunc destroy;

        char priv[0];
};

static inline Ret locker_lock(Locker* thiz)
{
        return_val_if_fail(thiz != NULL && thiz->lock != NULL, RET_INVALID_PARAMS);

        return thiz->lock(thiz);
}

static inline Ret locker_unlock(Locker* thiz)
{
        return_val_if_fail(thiz != NULL && thiz->unlock != NULL, RET_INVALID_PARAMS);
```

```
        return thiz->unlock(thiz);
}

static inline void locker_destroy(Locker* thiz)
{
        return_if_fail(thiz != NULL && thiz->destroy != NULL);

        thiz->destroy(thiz);

        return;
}
```

这里要求：

(1) 所有数据都隐藏起来，对外只提供接口函数；

(2) 所有接口函数都用inline函数进行包装。

下面我们看看接口的一些不同定义方式。

(1) 带有公开的数据，用普通函数进行包装。如：

```
struct CharDriverState {
        void (*init)(struct CharDriverState *s);
        int (*chr_write)(struct CharDriverState *s, const uint8_t *buf, int len);
        void (*chr_update_read_handler)(struct CharDriverState *s);
        int (*chr_ioctl)(struct CharDriverState *s, int cmd, void *arg);
        IOEventHandler *chr_event;
        IOCanRWHandler *chr_can_read;
        IOReadHandler *chr_read;
        void *handler_opaque;
        void (*chr_send_event)(struct CharDriverState *chr, int event);
        void (*chr_close)(struct CharDriverState *chr);
        void (*chr_accept_input)(struct CharDriverState *chr);
        void *opaque;
        int focus;
        QEMUBH *bh;
        char *label;
        char *filename;
        TAILQ_ENTRY(CharDriverState) next;
};
```

(以上代码取自qemu-0.10.5/qemu-char.h。)

```
int qemu_chr_write(CharDriverState *s, const uint8_t *buf, int len)
{
        return s->chr_write(s, buf, len);
}

int qemu_chr_ioctl(CharDriverState *s, int cmd, void *arg)
{
        if (!s->chr_ioctl)
            return -ENOTSUP;
        return s->chr_ioctl(s, cmd, arg);
```

```
}

int qemu_chr_can_read(CharDriverState *s)
{
        if (!s->chr_can_read)
            return 0;
        return s->chr_can_read(s->handler_opaque);
}

void qemu_chr_read(CharDriverState *s, uint8_t *buf, int len)
{
        s->chr_read(s->handler_opaque, buf, len);
}
```

（以上代码取自qemu-0.10.5/qemu-char.c。）

这段代码是著名虚拟机qemu的字符设备驱动的接口定义。它包含了部分数据成员，对函数指针的包装用的是普通函数，而不是inline函数。

(2) 带有公开的数据，并且不提供inline函数。如：

```
struct pci_driver {
        struct list_head node;
        char *name;
        const struct pci_device_id *id_table;
    /* must be non-NULL for probe to be called */
        int  (*probe)  (struct pci_dev *dev, const struct pci_device_id *id);
    /* New device inserted */
        void (*remove) (struct pci_dev *dev);
    /* Device removed (NULL if not a hot-plug capable driver) */
        int  (*suspend) (struct pci_dev *dev, pm_message_t state);
    /* Device suspended */
        int  (*suspend_late) (struct pci_dev *dev, pm_message_t state);
        int  (*resume_early) (struct pci_dev *dev);
        int  (*resume) (struct pci_dev *dev);                    /* Device woken up */
        void (*shutdown) (struct pci_dev *dev);
        struct pci_error_handlers *err_handler;
        struct device_driver    driver;
        struct pci_dynids dynids;
};
```

（以上代码取自linux-2.6.30/include/linux/pci.h。）

这是Linux内核对PCI驱动程序的接口定义，里面有部分公开的数据成员，而且没有提供inline函数的包装。虽然我不太喜欢这种做法，但出于性能和方便等方面考虑，这样做也是有道理的。这就是真实的世界，对的做法通常不只一种，我们必须学习从不同的角度来考虑问题。

(3) 把接口函数放在一个独立的结构中。如：

```
struct file {
        /*
     * fu_list becomes invalid after file_free is called and queued via
     * fu_rcuhead for RCU freeing
```

```
     */
        union {
            struct list_head    fu_list;
            struct rcu_head     fu_rcuhead;
        } f_u;
        struct path     f_path;
#define f_dentry    f_path.dentry
#define f_vfsmnt    f_path.mnt
    const struct file_operations    *f_op;
        spinlock_t      f_lock;  /* f_ep_links, f_flags, no IRQ */
        atomic_long_t       f_count;
        unsigned int        f_flags;
        fmode_t         f_mode;
        loff_t          f_pos;
        struct fown_struct  f_owner;
        const struct cred   *f_cred;
        struct file_ra_state    f_ra;

        u64         f_version;
#ifdef CONFIG_SECURITY
        void            *f_security;
#endif
        /* needed for tty driver, and maybe others */
        void            *private_data;

#ifdef CONFIG_EPOLL
        /* Used by fs/eventpoll.c to link all the hooks to this file */
        struct list_head    f_ep_links;
#endif /* #ifdef CONFIG_EPOLL */
        struct address_space    *f_mapping;
#ifdef CONFIG_DEBUG_WRITECOUNT
        unsigned long f_mnt_write_state;
#endif
};
```

（以上代码取自linux-2.6.30/include/linux/fs.h。）

这是Linux内核对文件的定义。我们知道在类Unix的系统中，文件不只是存放磁盘上的文件，还可以是硬件设备和虚拟设备的抽象。用户空间的程序调用read、write和ioctl等函数去操作文件，文件的这些接口函数是统一的，但不同类型文件的读写函数的实现是不同的。同样的功能有不同的实现，这就要用到接口了，所以文件是个接口。但从上面file的定义中，我们没有看到它的接口函数，原因是这些接口函数放在一个独立的结构file_operations里了。

```
struct file_operations {
        struct module *owner;
        loff_t (*llseek) (struct file *, loff_t, int);
        ssize_t (*read) (struct file *, char __user *, size_t, loff_t *);
        ssize_t (*write) (struct file *, const char __user *, size_t, loff_t *);
        ssize_t (*aio_read) (struct kiocb *, const struct iovec *, unsigned long, loff_t);
        ssize_t (*aio_write) (struct kiocb *, const struct iovec *, unsigned long, loff_t);
        int (*readdir) (struct file *, void *, filldir_t);
        unsigned int (*poll) (struct file *, struct poll_table_struct *);
```

```
        int (*ioctl) (struct inode *, struct file *, unsigned int, unsigned long);
        long (*unlocked_ioctl) (struct file *, unsigned int, unsigned long);
        long (*compat_ioctl) (struct file *, unsigned int, unsigned long);
        int (*mmap) (struct file *, struct vm_area_struct *);
        int (*open) (struct inode *, struct file *);
        int (*flush) (struct file *, fl_owner_t id);
        int (*release) (struct inode *, struct file *);
        int (*fsync) (struct file *, struct dentry *, int datasync);
        int (*aio_fsync) (struct kiocb *, int datasync);
        int (*fasync) (int, struct file *, int);
        int (*lock) (struct file *, int, struct file_lock *);
        ssize_t (*sendpage) (struct file *, struct page *, int, size_t, loff_t *, int);
        unsigned long (*get_unmapped_area)(struct file *, unsigned long, unsigned long,
unsigned long, unsigned long);
        int (*check_flags)(int);
        int (*flock) (struct file *, int, struct file_lock *);
        ssize_t (*splice_write)(struct pipe_inode_info *, struct file *, loff_t *,
size_t, unsigned int);
        ssize_t (*splice_read)(struct file *, loff_t *, struct pipe_inode_info *,
size_t,     unsigned int);
        int (*setlease)(struct file *, long, struct file_lock **);
    };
    (以上代码linux-2.6.30/include/linux/fs.h。)
```

这样做的好处是可以节省一些空间。file_operations中有25个函数指针，在32位的系统上，它会占用25×4=100字节。如果把file_operations里的接口函数直接放在结构file中，那对于每个文件对象都会占用100字节，假设系统中打开了1000个文件，那就会占用将近100KB的内存空间。

而采用上面的做法，结构file中只是保存一个file_operations的指针，那对于每个文件对象就只需要4个字节了。当然，这种方法使用起来要麻烦一点，速度也会慢一点（其实这都不是什么大问题了）。如果接口函数很多，而且同时存在的对象也很多，采用这种方法是可取的。

在C语言里，不管采用哪种形式定义接口，它们的本质都是一样的。都是通过函数指针来抽象具体的实现，不同之处主要是在方便性和封装性之间做些折中。在正常情况下，最好还是使用我们前面所学习过的方法。